Titre: Title:	Intégration d'un RTOS dans une plate-forme SystemC destinée à l'exploration architecturale
Auteur: Author:	Mathieu Rondonneau
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Rondonneau, M. (2004). Intégration d'un RTOS dans une plate-forme SystemC destinée à l'exploration architecturale [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7298/

UNIVERSITÉ DE MONTRÉAL

INTÉGRATION D'UN RTOS DANS UNE PLATE-FORME SYSTEMC DESTINÉE À L'EXPLORATION ARCHITECTURALE

MATHIEU RONDONNEAU
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JANVIER 2004

Your file Votre référence
ISBN: 0-612-90858-5
Our file Notre référence
ISBN: 0-612-90858-5

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

INTÉGRATION D'UN RTOS DANS UNE PLATE-FORME SYSTEMC DESTINÉE À L'EXPLORATION ARCHITECTURALE

présenté par: RONDONNEAU Mathieu

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

Mme. NICOLESCU Gabriela, Doctorat, président

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. DAGENAIS Michel, Ph.D., membre

REMERCIEMENTS

Je tiens à exprimer ma reconnaissance à Guy Bois, mon directeur de recherche. Je le remercie tout particulièrement pour son soutien, sa confiance ainsi que la flexibilité dont il a su faire preuve. Grâce à lui, ma maîtrise a été une expérience très enrichissante, tant sur le plan humain que professionnel.

Je remercie également Olivier Benny et Jérôme Chevalier constituant les membres de l'équipe du projet SPACE pour la motivation, la détermination et la bonne humeur dont ils ont fait preuve et qui a contribué à un environnement de travail agréable. Ils ont également contribué à compléter mes connaissances.

Je tiens également à remercier toutes les personnes qui ont participé à ce projet en apportant leur expérience, en particulier messieurs Mostapha Aboulhamid et François R. Boyer.

Je remercie sincèrement tous les étudiants du groupe CIRCUS qui ont contribué à mon intégration au sein d'une équipe dynamique.

Enfin, je tiens à remercier mes parents ainsi que ma soeur pour leurs encouragements et leur immense soutien depuis la France.

RÉSUMÉ

La bibliothèque SystemC fait parti des langages de description de systèmes hétérogènes à haut niveau les plus prometteurs. Cependant, des lacunes au niveau de la simulation logicielle restent problématiques pour la phase d'intégration. La méthodologie du présent travail propose donc un processus de raffinement logiciel basé sur SystemC. Ce raffinement est composé de trois niveaux d'abstraction en partant d'un niveau fonctionnel à haut niveau proche de la synthèse des systèmes sur puce. L'objectif est de permettre l'exploration architecturale afin de déterminer le meilleur compromis logiciel/matériel qui satisfait les spécifications.

La phase de spécification débute au premier niveau (niveau 1) où le système est décrit entièrement en SystemC. Ensuite, pour chacun des niveaux suivants (niveau 2 et 3), les modules utilisateurs peuvent être déplacés de la partie logicielle à la partie matérielle (ou inversement), sans modification du code de l'application. Au niveau 2, le système est partitionné. Le simulateur de SystemC ordonnance les modules matériels alors qu'un RTOS, encapsulé dans une API SystemC, ordonnance les modules logiciels. Enfin, au niveau 3, les modules sont connectés à la plate-forme SPACE incluant le même RTOS exécuté par un émulateur de processeur connecté à l'architecture et simulé par SystemC.

L'intégration d'une plate-forme SystemC et d'un RTOS permet la validation fonctionnelle en modélisant le comportement du système final. La méthodologie de raffinement permet aux développeurs de valider le fonctionnement du système à chacun des niveaux d'abstraction qui produisent des résultats de simulation de plus en plus précis.

ABSTRACT

SystemC is today the leading language for system-level modelling in C++. The lack of high-level software modelling may cause fastidious design integration. The methodology proposes a top-down approach divided in three levels to perform software refinement using SystemC 2.0 from a purely functional level towards system-on-chip synthesis.

These properties allow architectural exploration leading to integration problems avoidance. The specification process starts at level 1 with the use of a common language (SystemC). Thus, at any time of level 2 and level 3 the user modules can be moved to the software or hardware parts (and vice versa) of the architecture to enable effortless exploration, without having to modify the application code. At level 2, the application is partitioned in two parts. The SystemC simulator schedules the hardware modules, while an RTOS (Real-Time Operating System) emulation process encapsulated in a SystemC API can schedule the software modules. Then, at level 3, each partition can be connected to a platform named SPACE that includes the same RTOS, but here executed by a cycle-accurate ISS (Instruction Set Simulator) connected to the platform and scheduled by the SystemC simulator.

The integration of a target platform model and a RTOS scheduler in a SystemC simulation is a viable solution that focuses on application functional verification by modelling the intended behaviour of the whole system. The software refinement methodology lets system designers validate their application functionality at each abstraction level that provides more and more precise simulation results.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES NOTATIONS ET DES SYMBOLES

API. . . . . Application Programmable Interface

ASIC. . . . . Application Specific Integrated Circuit ou circuit intégré dédié

CPU. . . . . Central Processing Unit ou unité centrale

FIFO. . . . . First In First Out

FPGA. . . . . Field Programming Gate Array

HDL. . . . . Hardware Description Langage

IP. . . . . Intellectual Property

ISS. . . . . Instruction Set Simulator

OO. . . . . Object Oriented ou orienté objet

OSCI. . . . . Open SystemC Initiative

RAM. . . . . Random Access Memory

ROM. . . . . Read Only Memory

RPC. . . . . Remote Procedure Call

RTL. . . . . Register Transfer Level

RTOS. . . . . Real-Time Operation System

SoC. . . . . System-on-Chip

STL. . . . . Standard Template Library

SPACE. . SystemC Partitioning Architectures for Co-design of Embedded system

UML. . . . . Unified Modeling Langage

UT. . . . . Unités de Temp

UTF. . . . . Untimed Functional

VHDL. . . . . VHSIC Hardware Description Langage

VHSIC. . . . . Very High Speed Integrated Circuit

LISTE DES ANNEXES

INTRODUCTION

0.1 Historique

Les premiers systèmes embarqués sont apparus en 1971 avec l'apparition du Intel 4004. Développé en 1971, ce premier microprocesseur, a été le premier circuit intégré incorporant tous les éléments d'un ordinateur dans un seul boîtier : unité de calcul, mémoire et contrôle d'entrées / sorties.

Alors qu'il fallait auparavant plusieurs circuits intégrés différents, chacun dédié à une tâche particulière, un seul microprocesseur pouvait assurer autant de travaux différents que possible. Très rapidement, des objets quotidiens tels que fours à micro-ondes, télévisions et automobiles à moteur à injection électronique ne tardèrent pas à être équipés de microprocesseurs.

Ce fut là les débuts de l'informatique embarquée.

0.2 Qu'est ce qu'un système embarqué

Un système embarqué est un système autonome composé d'une partie électronique (généralement un circuit intégré) et une partie logicielle (un programme résidant en mémoire). Il est défini aussi généralement par le fait qu'il n'est pas visible en tant que tel, mais est intégré dans un équipement doté d'une autre fonction.

Voici les systèmes informatiques embarqués que nous utilisons quotidiennement : gestion de l'ascenseur, auto radio, calculateur pour la sécurité automobile (coussin gonflable), distributeur de boissons, routeur Internet, téléphone mobile, distributeur de billets, console de jeux.

0.3 Caractéristiques

Du fait de la nécessité d'une architecture physique confinée, les parties matérielles et logicielles d'un système embarqué sont intimement liées. Elles ne sont pas aussi discernables que dans un environnement classique de type PC. Par ailleurs, la conception de ces systèmes est généralement fiable (aéronautique, système de sécurité automobile) à cause de leur utilisation dans des domaines à fortes contraintes dont dépend la sécurité des hommes, mais également parce que l'accès au logiciel est souvent difficile et coûteux, une fois le système fabriqué.

De nombreuses applications des systèmes embarqués sont des applications temps réel. En effet, il y a une faible frontière entre les systèmes embarqués et les systèmes temps réel. Cependant, un logiciel embarqué qui exploite des ressources limitées et qui réalise certaines fonctionnalités en relation avec l'extérieur, n'a pas forcément de contraintes temps réel.

Un système est qualifié de temps réel lorsqu'il est capable d'interagir avec un environnement externe qui lui-même évolue avec le temps. Un système temps réel rencontre donc des contraintes temporelles. Ses sorties doivent être présentées au bon moment. C'est à dire que l'information après acquisition et traitement reste encore pertinente. Dans le cas d'une information arrivant de façon régulière (sous forme d'une interruption périodique du système), le temps d'acquisition et de traitement doivent rester inférieurs à la période de rafraîchissement de cette information. Il existe plusieurs niveaux de contraintes temporelles qui permettent de caractériser les systèmes temps réel :

- Les systèmes à contraintes souples ou molles (ou en anglais : *soft real time*) : La performance de ses systèmes est dégradée mais sans engendrer des conséquences dramatiques si les contraintes temporelles ne sont pas rencontrées. Ces systèmes acceptent des variations dans le traitement des données de l'ordre des 500 ms ou

de la seconde. Ce sont par exemple des systèmes multimédia : si quelques images ne sont pas affichées, cela ne met pas en péril le fonctionnement correct de l'ensemble du système. Ces systèmes se rapprochent fortement des systèmes d'exploitation classiques à temps partagé. Ils garantissent un temps moyen d'exécution pour chaque tâche. La répartition du temps CPU est équilibrée entre les processus.

- Les systèmes à contraintes dures (ou en anglais : *hard real time*) : Leur incapacité de rencontrer les contraintes temporelles cause la faute du système. Une gestion stricte du temps est nécessaire pour conserver l'intégrité du service rendu. Ce sont par exemple les contrôles de processus industriels sensibles comme la régulation des centrales nucléaires ou les systèmes embarqués utilisés dans l'aéronautique. Ces systèmes garantissent un temps maximum d'exécution pour chaque tâche.

Les systèmes à contraintes dures doivent répondre à trois critères fondamentaux :

1. Le déterminisme logique : les mêmes entrées appliquées au système doivent produire les même effets.
2. Le déterminisme temporel : une tâche donnée doit obligatoirement être exécutée dans un délai imparti, la notion d'échéance est prise en compte.
3. La fiabilité : Le système doit être disponible. Cette contrainte est très forte dans le cas d'un système embarqué car les interventions d'un opérateur sont très difficiles voire même impossibles. Cette contrainte est indépendante de la notion de temps réel mais la fiabilité du système sera d'autant plus mise à l'épreuve dans le cas de contraintes dures.

En résumé, on peut dire qu'un système temps réel doit être prévisible, les contraintes temporelles pouvant s'échelonner entre quelques micro-secondes (μs) et quelques secondes. Les systèmes embarqués temps réel requièrent la présence d'un mécanisme d'ordonnancement adapté au temps réel (le plus connu est celui basé sur la priorité des tâches). Il arrive parfois que ce soit uniquement des systèmes dédiés à une

tâche, dans ce cas, il n'y a pas de mécanisme d'ordonnancement, le système exécute indéfiniment une seule et même boucle (circuit fermé).

0.4 Architecture d'un système embarqué

Cette architecture peut varier selon les systèmes : certains composants jugés non pertinents pour la fonction à réaliser, peuvent être inexistants. Par exemple, le co-processeur n'est pas indispensable dans les nombreux systèmes embarqués autonomes et indépendants. En revanche, l'architecture de base est la plupart du temps, composée d'une unité centrale de traitement (CPU), d'un système d'exploitation qui réside parfois uniquement en un logiciel spécifique (routeur réseaux), ou une boucle d'exécution (mécanisme du coussin gonflable). De même, l'interface utilisateur (Interface Homme Machine) n'est pas souvent existante, mais reste utile pour configurer le système, vérifier son comportement ou, dans la plupart des autres cas, permettre une requête formulée par l'utilisateur, et y répondre.

La figure 0.1 présente une architecture d'un système embarqué.

Gardons à l'esprit qu'un système embarqué est composé d'une partie logicielle et d'une autre matérielle dont les caractéristiques dépendent de ses spécifications et de ce pour quoi il est dédié. Il existe donc un grand nombre de systèmes embarqués différents, dépendamment des fonctionnalités recherchées.

Le fonctionnement du système présenté à la figure 0.1 se résume ainsi :

- L'acquisition : L'objectif est de recevoir des informations de l'environnement extérieur et de les convertir en signaux numériques (capteurs).

- Le traitement : L'objectif est de traiter les informations reçues. Cette tâche est effectuée par l'unité de traitement composée d'un ensemble de composants comme

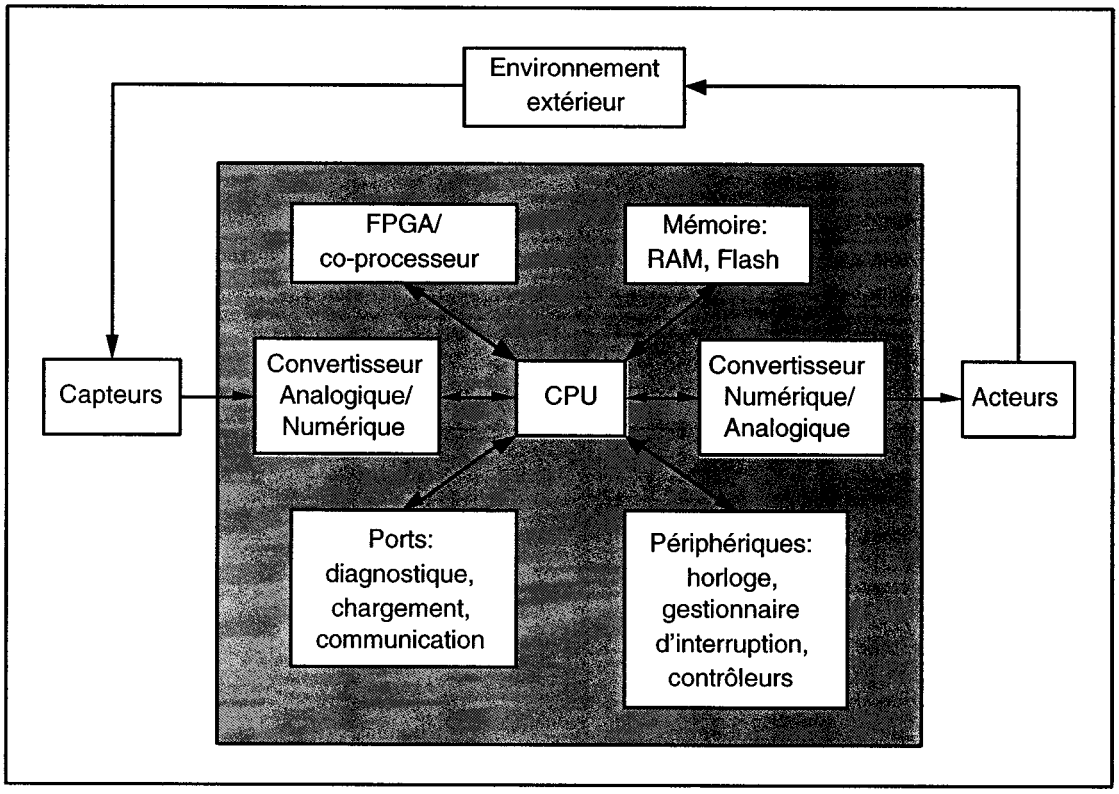

FIGURE 0.1 Architecture d'un système embarqué

par exemple : un CPU, de la mémoire, un programme logiciel, et éventuellement un FPGA (composant programmable) ainsi que des composants nécessaires au traitement des informations.

- la réaction : L'objectif est de modifier l'environnement extérieur dépendamment des résultats du traitement obtenu. Ceci peut se faire sous différentes formes par l'intermédiaire de périphériques, de contrôle de commandes ou d'interfaces utilisateurs (acteurs).

0.5 Le développement des systèmes embarqués

Le développement des systèmes embarqués n'est pas une tâche simple. Celle-ci renferme en effet une multitude de contraintes dont les plus importantes sont la surface, le coût et les performances. Ces contraintes sont fortement liées. En effet, le logiciel est plus facile à maintenir, quant au matériel, il est plus performant mais plus coûteux. Il y a donc un compromis à définir, lors du développement, entre les parties du système qui seront matérielles et celles qui seront logicielles.

Évidemment, il existe également d'autres contraintes que l'on peut catégoriser comme étant liées à l'environnement final : consommation, dissipation de puissance, évolution, performance.

Nous venons de voir que les performances sont régies par les contraintes de taille. Cependant, cette contrainte tend à disparaître du fait de l'intégration des composants électroniques (augmentation du nombre de transistors). Malheureusement, de nouvelles contraintes apparaissent, comme la puissance dissipée et les coûts du fait de cette intégration.

Gardons également à l'esprit qu'un système temps reel n'est pas forcément plus rapide qu'un système classique. Il devra par contre satisfaire à des contraintes temporelles strictes, prévues à l'avance et imposées par le processus extérieur à contrôler. Une confusion classique est de mélanger temps réel et rapidité de calcul du système donc puissance du processeur (microprocesseur, micro-contrôleur, DSP). En effet, rappelons qu'un système est caractérisé comme étant temps réel à condition d'être capable d'acquitter l'interruption périodique (moyennant un temps de latence d'acquittement d'interruption imposé par le matériel), de traiter l'information et le signaler dans un temps inférieur au temps entre deux interruptions périodiques consécutives. Nous sommes donc lié à la contrainte de délai entre deux

interruptions générées par le processus extérieur à considérer.

Dans le cas où cette durée est de l'ordre de la seconde (pour contrôle d'une réaction chimique par exemple), il ne sert à rien d'avoir un système à base de Pentium IV. Un simple processeur 8 bits du type micro-contrôleur Motorola 68HC11, Microchip PIC ou même un processeur 4 bits fera amplement l'affaire, ce qui permettra de minimiser les coûts de fabrication.

Si ce temps est maintenant de quelques dizaines de microsecondes (pour le traitement de données issues de l'observation d'une réaction nucléaire par exemple), il convient de choisir un processeur nettement plus performant comme un processeur 32 bits Intel x86, StrongARM ou Motorola ColdFire.

Il convient donc avant de concevoir le système de connaître la durée minimale entre deux interruptions, ce qui est assez difficile à estimer voir même impossible. C'est pour cela que l'on a tendance à concevoir dans ce cas des systèmes performants (en terme de puissance de calcul CPU et de rapidité de traitement d'une interruption) et souvent sur-dimmensionnés pour respecter des contraintes temps réel mal cernés à priori. Ce sur-dimmensionnement implique un coût non négligeable.

0.6 Les outils

L'utilisation d'outils de génie logiciel est fortement conseillée pour le développement de systèmes embarqués. Le logiciel (le programme) est en général destiné à être intégré dans un composant produit de manière industrielle (à très grand nombre d'unités) où les mises à jours, quand elles sont possibles, ne sont pas triviales et renferment un coût relatif (moins complexe et moins coûteux qu'une mise à jour ou modification au niveau matériel). Le développement est par conséquent très soigné, l'accent est mis sur la fiabilité.

Dès les spécifications, des outils de modélisation sont utilisés. Parmi eux, nous pouvons distinguer l'apparition des langages de description de systèmes à haut niveau. Nous reviendrons plus en détail au cours des prochains chapitres concernant le type de langage qui a été utilisé dans ce mémoire.

0.7 L'intégration

L'évolution très rapide des technologies de fabrication de circuits intégrés sur silicium permet déjà de réaliser des systèmes complets (logiciel et matériel) intégrés sur une même puce (System On Chip). Cette intégration permet de concevoir des systèmes de plus en plus complexes (1 milliard de transistors sur une puce à l'horizon 2010). Afin de réduire le temps de développement, des blocs fonctionnels appelés généralement IP (pour Intellectual Property) sont de plus en plus utilisés. Leur fonctionnalité peut correspondre à un processeur ou même à un ordinateur complet avec sa mémoire et ses entrées/sorties.

De ce fait, les outils classiques de conception assistée par ordinateur dans les domaines de la microélectronique doivent évoluer en prenant en compte les aspects des systèmes hétérogènes. Le logiciel et le matériel se partagent la fonctionnalité du système de manière étroite, puisqu'ils cohabitent ensembles. Il s'agit d'une modification totale de la méthodologie de développement pour permettre la conception, l'intégration et la validation du système final en prenant en compte les contraintes fixés par les spécifications et celles de la communication entre les deux mondes (logiciel et matériel).

Ces systèmes sur puce sont déjà présents dans les objets communicants actuels (téléphones portables, agendas électroniques). Ils concernent donc des enjeux sociaux et économiques cruciaux du fait de leur présence dans la totalité des équi-

pements destinés aux technologies de l'information et de la communication. Le Groupe de Recherche en Microélectronique (GRM) contribue à ce domaine de par ses différents projets. Son objectif est de proposer de nouvelles méthodologies de conception pour systèmes embarqués et systèmes sur puce (SoC ou System on Chip). Ce domaine d'étude est aussi appelé *codesign* logiciel/matériel dans la littérature.

0.8 Problématique

Nous avons vu que l'évolution des systèmes sur puce se traduit par une forte demande d'outils et de méthodes permettant la conception, l'intégration et la validation des composants hétérogènes. Dans un premier temps, un langage décrivant à la fois les fonctionnalités matérielles et logicielles d'un système est donc nécessaire. En plus d'une simple description fonctionnelle haut niveau, il pourrait permettre un raffinement jusqu'à l'implémentation matérielle et logicielle. Des librairies pour la modélisation matérielle sont de plus en plus intéressantes du fait de l'augmentation de la complexité des systèmes embarqués. Aujourd'hui SystemC a pris une place de choix dans la modélisation de systèmes hétérogènes en C++ (langage orienté objet). Nous verrons que la bibliothèque SystemC propose un raffinement progressif des spécifications. Le cycle de modélisation débute par un système abstrait à haut niveau avec contrainte de temps ou non. Pour la partie matérielle, le système est ensuite raffiné au niveau cycles, puis au niveau RTL (Register Transfer Level). SystemC n'est pas une méthodologie de modélisation, mais sa force, pour la modélisation matérielle, réside dans sa variété de niveaux d'abstraction permettant la modélisation du système très tôt dans le cycle de développement. Cette méthodologie ne propose pas de solution concrète pour le partitionnement, la description du logiciel, les liens avec le système d'exploitation et la communication

entre les parties matérielles et logicielles. Une des lacunes de SystemC, du côté de la modélisation de la partie logicielle, est l'ordonnancement des processus logiciels. En effet, le simulateur utilise le même ordonnancement pour le logiciel que pour le matériel. Afin d'obtenir le même comportement utilisé pour les contraintes temps réel, c'est à dire la gestion des interruptions ainsi que la préemption, l'intégration d'un véritable ordonnanceur (système d'exploitation temps réel) serait un atout.

0.9 Méthodologie

L'utilisation de langages de description à haut niveau concernant la partie logicielle reste encore difficile due à la complexité lors de la synthèse. En effet, le code de l'application doit, la plupart du temps, être modifié afin d'intégrer l'architecture finale. Toute modification de code lors de la phase d'intégration peut être source d'erreur et doit être évitée. La méthodologie proposée dans ce mémoire regroupe trois niveaux permettant un raffinement logiciel utilisant SystemC 2.0 à partir d'une description à très haut niveau jusqu'à un niveau très proche de la synthèse dans un système sur puce. Un premier niveau, appelé L1 (Level1) permet la spécification de l'application ainsi que sa validation fonctionnelle à l'aide du simulateur SystemC. Au deuxième niveau, appelé L2 (level2), l'application est partitionnée en deux parties : les modules logiciels et les modules matériels. La partie matérielle est exécutée avec le simulateur de SystemC. Tandis que la partie logicielle est ordonnancée par un RTOS exécuté comme étant un processus encapsulé dans une API SystemC. Enfin, au troisième niveau, appelé L3 (level3), chacune des partitions est connectée sur la plateforme SPACE, incluant le même RTOS s'exécutant sur un émulateur de processeur ARM (ISS) ordonnancé par le simulateur de SystemC. La contribution de la méthodologie réside dans le fait de permettre un partitionnement à chacun des trois niveaux d'abstraction. Cette caractéristique permet l'exploration archi-

tecturale ainsi que le raffinement logiciel dans le but de faciliter la phase délicate d'intégration.

0.10 Contribution et originalité

Ce document propose donc une méthodologie à plusieurs niveaux de raffinement pour la partie logicielle, s'intégrant dans un outil d'exploration architecturale. Elle vise à encapsuler un véritable RTOS dans une API SystemC permettant la simulation de la partie logicielle du système à haut niveau d'abstraction. Cette méthodologie fait partie d'un plus vaste projet de développement d'un outil d'aide au partitionnement des systèmes sur puce (ou SoC). Cet outil permettra la simulation d'un système décrit en SystemC composé de modules logiciels et matériels qui pourront être déplacés de la partie logicielle vers la partie matérielle ou inversement, de manière à satisfaire les contraintes et spécifications. Il vise également à réduire considérablement le temps de développement de ses systèmes sur puce.

0.11 Plan du document

Nous allons tout d'abord discuter des différents travaux actuels dans ce domaine. Puis, nous présenterons la méthodologie en expliquant les trois niveaux de raffinement logiciel, et nous détaillerons la partie logicielle s'intégrant dans le développement d'un outil de recherche architectural qui sera abordé dans un chapitre séparé. Puis, nous discuterons des premiers résultats obtenus. Enfin, la conclusion permettra d'identifier les limites et les contraintes, tout en mettant l'accent sur les contributions de ce projet ainsi que ses évolutions futures.

CHAPITRE 1

REVUE DE LITTÉRATURE

Actuellement, il n'existe aucun mécanisme permettant de déterminer de manière automatique la partie matérielle et la partie logicielle d'un futur système sur puce. En effet, le cycle de développement débute par une spécification du système. Ensuite, le partitionnement, c'est à dire l'attribution de la partie du système qui sera matérielle et celle qui sera logicielle, est déterminé arbitrairement par les concepteurs expérimentés. Ce choix est également guidé par les précédents développements favorisant ainsi la réutilisation qui réduit les coûts et le temps de développement. Cependant, cette méthode peut impliquer des erreurs de conception très difficiles à identifier au début du développement. En effet, les parties réutilisées ne sont pas forcément bien adaptées au nouveau système, de plus, le partitionnement logiciel/matériel choisi peut s'avérer inefficace, non conforme aux spécifications. Ses erreurs n'apparaissent que très tard dans le processus de développement, c'est à dire par exemple, lors de l'intégration. A ce niveau, la modification du partitionnement logiciel/matériel requiert un re-développement complet du système, retardant considérablement la date de mise en marché du produit final. La majeure partie de ces problèmes provient du fait de l'inexistence d'outils d'aide au partitionnement , facilitant ainsi la phase délicate d'intégration. Afin d'automatiser le partitionnement pour éviter toute erreur lors d'une intervention humaine, le système hétérogène doit être préalablement décrit dans un seul et même langage. Comme nous l'avons vu en introduction, le développement des SoCs est très complexe et requiert une modélisation dans le processus de développement dès la phase de spécifications par la description du système dans un langage de haut niveau.

1.1 Les langages de description haut niveau

Depuis quelques années, des bibliothèques permettant la conception système ont vu le jour. Elles permettent d'abstraire des éléments d'un système embarqué pour effectuer des simulations à un niveau le plus élevé possible. Des outils très intéressants tels SpecC [11] et SystemC [25] remplissent cette fonction. En effet, basées sur une représentation à plusieurs niveaux d'abstractions d'un système hétérogène, ce sont de véritables bibliothèques C/C++. Nous allons parler plus particulièrement de SystemC puisque cette bibliothèque est relativement nouvelle par rapport à SpecC. De plus, celle-ci est au coeur du projet présenté dans ce mémoire. Enfin, régie par l'OSCI (regroupement d'entreprises et d'universités), elle est en voie de devenir un standard.

1.1.1 SystemC

SystemC est une nouvelle méthodologie de modélisation de systèmes hétérogènes. Sa structure est fournie comme une bibliothèque de classes C++ permettant la modélisation ainsi que la simulation à différents niveaux d'abstraction d'un système hétérogène (composé à la fois de composants logiciels et matériels). Son code source est complètement disponible.

La philosophie de SystemC est très intéressante puisqu'elle propose plusieurs niveaux de représentation : de la description haut niveau vers la synthèse d'un système hétérogène (logiciel/matériel) en utilisant un seul et même langage, le C++. Du fait de sa popularité relativement récente, elle ne peut couvrir l'étendu des caractéristiques logicielles et matérielles, et comporte encore quelques fonctions non disponibles au niveau de la simulation, mais son avenir reste toutefois très prometteur.

1.1.2 Le langage C++

Ce langage, aujourd'hui l'un des plus répandu est né d'une encapsulation du langage C. C'est à dire qu'il conserve ses principaux avantages en corrigeant ses défauts et en rajoutant des concepts plus évolués pour en faire un langage complexe et très puissant. De plus, il conserve une vitesse d'exécution relativement identique à son prédécesseur. Le second avantage est sa modularité. Il permet de structurer la conception d'une application en modules (classes) qui peuvent être développés indépendamment. Par la suite, la modification d'un module reste généralement transparente pour les autres, favorisant ainsi la réutilisation. Grâce à ses avantages, le C++ est un langage très apprécié pour les grands projets, et permet un développement efficace, rapide, et facile à maintenir. Celui-ci est donc un véritable atout pour la bibliothèque SystemC.

1.1.3 Historique

Comme nous l'avons expliqué précédemment, la bibliothèque SystemC est relativement récente dans le milieu universitaire et industriel. Elle est actuellement disponible en version 2.0.1. Voici les évolutions majeures depuis sa création.

- 1999 : SystemC 1.0. Cette première version a eu pour objectif de rivaliser avec les langages de description matérielle comme VHDL [30] et Verilog [29]. En effet, elle fournit un ensemble d'éléments permettant de modéliser et simuler des composants matériels avec le langage C++.
- 2001 : SystemC 2.0.1 : Cette nouvelle version laisse place à une nouvelle construction grâce à des classes abstraites permettant l'utilisation de canaux de communication [15]. Notons que ce concept a été emprunté à la bibliothèque SpecC. Cette bibliothèque permet de décrire le système logiciel/matériel à plus haut niveau

d'abstraction. En effet, elle permet par exemple la modélisation de SoCs (System on Chip) comportant à la fois des fonctionnalités logicielles et matérielles à haut niveau. Cette version permet également un raffinement des différents blocs constituant le système, qu'ils soient matériels ou logiciels, en proposant plusieurs niveaux d'abstraction. Ces niveaux permettent une simulation fonctionnelle mais, également une simulation plus précise, basée sur le déroulement du système au niveau cycle d'horloge.

La bibliothèque SystemC comporte encore quelques lacunes au niveau de la simulation logicielle. En effet le simulateur ne renferme aucun mécanisme d'ordonnancement capable de représenter fidèlement le comportement de processus logiciels. Les interruptions ainsi que la préemption, disponibles par les RTOS commerciaux ne sont pas, pour le moment, gérés par le simulateur de SystemC. Actuellement, la bibliothèque ne propose pas de modélisation des RTOS. Cependant, elle est toujours en constante évolution et une prochaine version 3.0 est prévue pour l'année 2004.

Les caractéristiques de SystemC seront décrites plus en détail dans le chapitre suivant (SystemC et la simulation logicielle).

1.2 Les plates-formes de développement

Il est donc possible de modéliser à haut niveau une application composée à la fois d'une partie logicielle et d'une partie matérielle. Nous venons de voir que des langages de description au niveau système existent. L'étape suivante est donc l'intégration des méthodologies pour raffinement utilisant les plates-formes hétérogènes modélisées à haut niveau.

Dans le monde industriel, plusieurs outils sont disponibles. N2C Design System

de CoWare [7] propose une méthodologie et un environnement de développement haut niveau. L'objectif est de fournir une plate-forme de base, équipée d'un processeur, d'un bus de communication et d'un gestionnaire de mémoire, permettant d'accueillir des fonctionnalités matérielles supplémentaires comme par exemple des IP (Intellectual Property). Les modules matériels se lient à plusieurs interfaces implantant des protocoles de communication à différents niveaux d'abstraction. Les interfaces matérielles seront générées automatiquement. Cette plate-forme permet, grâce aux outils et à la méthodologie, la configuration (ajout de composants matériels et logiciels) d'une plate-forme de base générique à haut niveau. Elle permet également de manière automatique la génération du code de la partie logicielle et matérielle pour l'intégration finale. L'environnement de simulation permet de fournir également des résultats qui peuvent amener à reconsidérer par exemple, le choix du partitionnement. Cet outil commercial n'a eu que très peu de succès peut être à cause de son lancement prématuré.

Des outils de co-simulation tels que Seamless CVE [22] permettent la simulation du système après partitionnement. L'environnement permet en effet la simulation de la partie logicielle décrite en C/C++ utilisant les ressources de la partie matérielle décrite en VHDL. Malheureusement, ceci diminue l'espace de recherche au niveau architectural. L'outil VCC Cierto [2],[28] de Cadence présente une méthodologie pour la conception des SoC en utilisant une description à haut niveau. Les spécifications sont décrites en C++, à l'aide d'une bibliothèque liée à l'outil. L'objectif est d'utiliser des blocs prédéfinis (IP) afin d'augmenter la productivité puis de simuler le tout au niveau fonctionnel. La spécification est liée à une plate-forme et un RTOS abstrait, entièrement configurables et sélectionnés à partir d'une bibliothèque. Cette bibliothèque est composée des modèles architecturaux et de performance pour de nombreux processeurs, contrôleurs, DSP, bus et mémoire. Ensuite, les résultats de simulation du système permettent d'effectuer d'éventuelles

corrections. La méthodologie offre l'environnement idéal pour l'exploration architecturale à haut niveau. Elle reste toutefois peu intuitive et, l'outil est encore très complexe.

1.3 Le raffinement logiciel

La prochaine étape après la modélisation à haut niveau est donc le raffinement. Nous parlons de raffinement pour désigner le passage d'un niveau d'abstraction élevé vers un niveau d'abstraction proche de l'intégration dans le système final.

Le projet suivant [17] présente une méthode permettant la génération automatique du code logiciel d'une plate-forme de co-design, développée en SystemC. L'objectif est de conserver le code SystemC de l'application tout au long du cycle de développement : de la spécification jusqu'à la génération du code logiciel pour la synthèse en passant par le partitionnement et la co-simulation.

Une fois l'application décrite et exécutée avec le simulateur SystemC pour être validée, celle-ci est partitionnée. La partie matérielle est encore simulée par le simulateur SystemC tandis que la partie logicielle est ordonnancée par un RTOS commercial à l'aide d'une bibliothèque appelée SC2RTOS. Celle-ci permet la redéfinition des éléments de la bibliothèque SystemC en appels des services d'un véritable RTOS. Ainsi, les mécanismes de concurrence et de communication de SystemC sont remplacés par la gestion multitâche d'un RTOS. Cette approche est indépendante du RTOS utilisé, permettant ainsi au concepteur de choisir le RTOS satisfaisant les spécifications.

La méthodologie proposée ici est très proche de celle décrite dans ce mémoire. Néanmoins elle ne s'intègre pas dans un processus d'aide au partitionnement logiciel/matériel puisqu'elle ne permet pas la remise en cause du choix du partitionne-

ment dans le processus de développement.

Dans le cas où les résultats de simulation ne satisferaient pas les spécifications, le partitionnement doit être remis en question, ce qui implique une modification directement dans le code de l'application et tend donc à rallonger le temps de développement.

Des outils permettant le passage d'une description haut niveau à un squelette d'un code synthétisable existent. Cependant, l'intervention humaine est encore nécessaire pour compléter le code qui peut être généré par des outils de manière automatique. Le passage d'un très haut niveau vers la synthèse est un problème très complexe. Une solution serait d'utiliser plusieurs niveaux d'abstraction. Chacun des niveaux permettrait de valider les fonctionnalités et contraintes de l'application en ajoutant une spécification de l'architecture finale de manière à tendre petit à petit vers la phase finale qui est la synthèse.

Une méthodologie [24] propose un processus de validation à plusieurs niveaux d'abstraction pour les parties logicielles et matérielles. La partie matérielle de l'application est composée de deux niveaux : le niveau comportemental où seulement la fonctionnalité du système est valide, et le niveau RTL (Register Transfert Level) dans lequel l'implémentation matérielle ainsi que les protocoles de communication sont déterminés près à être implantés. La partie logicielle propose trois niveaux d'abstraction. Le niveau supérieur permet d'abstraire le RTOS. L'ordonnancement de l'application est alors effectué par l'environnement de simulation. Ensuite, un niveau intermédiaire permet d'abstraire les caractéristiques matérielles de l'architecture finale. Les services du RTOS sont implémentés, l'application peut être validée avec le véritable comportement du RTOS. Enfin, le dernier niveau permet la validation finale avant l'intégration dans l'architecture. Rappelons que la méthodologie présentée ici est destinée à la validation et non à l'exploration architecturale.

1.4 La modélisation des RTOS pour la simulation

L'intégration de fonctionnalités logicielles dans les langages de description à haut niveau représente une des prochaines étapes dans l'évolution d'un tel outil de modélisation et de simulation. Nous allons dans un premier temps, nous attarder plus en détails sur les travaux effectués dans le domaine de la simulation de RTOS à haut niveau.

L'objectif de la modélisation d'un RTOS consiste à émuler le fonctionnement du système d'exploitation final. Ceci a pour objectif d'obtenir une représentation fonctionnelle de l'application avant même l'intégration, c'est à dire très tôt dans le cycle de développement.

Il existe actuellement plusieurs solutions proposant la modélisation de RTOS. Certains outils fournissent en effet un environnement complet permettant d'automatiser le développement de RTOS correspondant aux fonctionnalités nécessaires pour le fonctionnement de l'application. Parmi eux, nous distinguons :

- CarbonKernel [19] véritable simulateur basé sur les événements, reproduisant ainsi le comportement d'un système d'exploitation temps réel pour l'implémentation et les tests de l'application.

- SoCOS [9] est un modèle haut niveau de RTOS basé sur la génération de code logiciel en fonction de l'application.

- OsKit [27] une librairie composée de 34 composants disponibles sous forme de modules permettant de concevoir un système d'exploitation à la carte.

Entièrement paramétrables, ces outils permettent une simulation rapide de l'application en fournissant les principales fonctionnalités d'un véritable système d'exploitation commercial.

Cependant, le code du système d'exploitation obtenu par ses outils étant différent du code du système d'exploitation final, une étape de validation supplémentaire reste nécessaire. En effet, le code de l'application utilise les appels de fonctions fournies par l'émulateur de RTOS qui sont différents de ceux du RTOS finale. Cela entraîne une intervention humaine, et peut ralentir le processus de validation. Enfin, l'évaluation des performances de temps est également difficile à obtenir du fait de l'utilisation d'un système d'exploitation différent, dans un environnement trop éloigné du système final.

Cependant, WindRiver [33] fournit un modèle de simulation pour son système d'exploitation temps réel VxWorks [32] du nom de Vxsim [31]. Partie intégrante de la chaîne de développement, ce simulateur utilise le même ordonnanceur que le noyau final. L'interface pour le développement d'applications est également identique. La gestion des interruptions est également possible par émulation en utilisant le mécanisme des signaux de linux. Il est cependant limité puisqu'il ne permet en aucun cas la simulation avec des composantes matérielles de la plate-forme finale et donc ne permet pas l'évaluation des communications et des performances en terme de temps entre la communication logicielle et matérielle.

L'utilisation de simulateurs de RTOS pour le développement a donc ses limites concernant la validation des applications hétérogènes comportant une partie matérielle et logicielle. L'enjeu suivant est donc l'intégration de ces techniques de simulation de RTOS dans les langages de description et de simulation incluant des composants logiciels et matériels.

De nombreuses réalisations ont été effectuées au niveau de la modélisation de RTOS dans l'objectif d'augmenter les capacités des outils de modélisation (tel que SystemC et SpecC). Une modélisation d'un système d'exploitation temps réel, a été réalisée en utilisant la bibliothèque SpecC [13]. Le travail réalisé montre comment

il est possible de modéliser le comportement d'un système multitâche à un niveau d'abstraction élevé comblant ainsi les limites des langages de description systèmes à haut niveau.

Le RTOS modélisé vient prendre place entre l'application et le langage de description au niveau (SpecC). En effet, celui-ci fournit à l'application les services d'un RTOS et utilise les primitives de SpecC ainsi que des fonctionnalités permettant la communication avec les composants matériels modélisés eux aussi à l'aide de la bibliothèque SpecC. Le RTOS modélisé ne fournit pas les fonctionnalités identiques d'un véritable RTOS au niveau de la préemption et de la gestion des interruptions. Cependant, les premiers résultats obtenus sont prometteurs puisque le modèle n'a que très peu d'impact sur le temps de simulation. Les résultats restent basés sur une modélisation et non sur un véritable RTOS et encore moins sur une plate-forme finale. De plus, l'application ainsi modélisée et testée fonctionnellement doit être recodée complètement pour être intégrée dans l'architecture finale.

Une méthodologie [23] propose un modèle de simulation natif pour la validation du code final du système d'exploitation. L'objectif est de compenser, lors de la simulation du système, la lenteur d'un émulateur de processeur (ISS) par la simulation native du RTOS en respectant sa structure et ses fonctionnalités, en fournissant des informations temporelles et enfin de permettre la génération automatique du modèle. La méthode utilisée repose sur l'utilisation du code final du RTOS et l'ajout d'annotations temporelles. Elle propose également un remplacement des services spécifiques au processeur cible par l'intermédiaire d'un modèle fonctionnel implémenté d'une manière spécifique à l'environnement de simulation (par exemple SystemC, SpecC). Ainsi, les modèle de simulation pour les services non dépendants du processeur peuvent être obtenus en y ajoutant des délais pour les appels système dépendants du processeur. Quant aux services dépendants du processeur, ceux-ci sont remplacés par un modèle de simulation en y ajoutant des

notations temporelles. Ceci permet d'obtenir un modèle de RTOS conservant les même caractéristiques d'exécution (en terme de temps) que sur l'architecture finale tout en diminuant le temps de simulation puisque le RTOS est exécuté sur une machine de développement et non par un ISS. La difficulté réside dans l'affectation des délais dans le modèle de simulation. Pour ce faire, des méthodes d'estimation du temps d'exécution du logiciel sont employées. L'obtention des temps d'exécution spécifiques au matériel et au processeur cible s'effectue par des tests directement sur la cible. Ces temps sont ensuite ajoutés en paramètre dans des fonctions de délais insérées dans le modèle du RTOS reproduisant ainsi le comportement du système d'exploitation au niveau des délai d'exécution des services.

Le travail suivant [10] présente une méthodologie permettant la modélisation d'un RTOS préemptif, basé sur la priorité. S'appuyant sur l'interface de SystemC, c'est un RTOS entier qui est recréé ici comportant un mécanisme d'interruptions, une gestion d'événements ainsi qu'une gestion pour l'ordonnancement des différentes tâches. Une API propre au RTOS permet à l'application d'interagir avec le RTOS, ainsi qu'avec le BFM (Bus Functional Model) communicant avec les périphériques de la partie matérielle. Le RTOS abstrait reste un modèle, il ne permet en aucun cas l'évaluation en terme de temps de l'ensemble du système. De plus, le raffinement logiciel est inexistant, la modification de l'application est donc inévitable. Le code de l'application devra utiliser les appels de fonctions du RTOS utilisé sur la plate-forme finale. De plus, dépendamment du RTOS utilisé, le code peut être amené à changer dépendamment de ses fonctionnalités qui pourraient amener à un comportement différent.

Toutes ces techniques de modélisation de RTOS sont très intéressantes puisqu'elles tendent à reproduire le comportement d'un véritable RTOS dans un langage de description à haut niveau permettant ainsi la co-simulation de composants logiciels et matériels. Cependant, le code de l'application de la partie logicielle doit

être modifié par le concepteur afin qu'il utilise les fonctions du véritable RTOS. Il n'existe aucune méthode permettant le raffinement de l'application modélisée à haut niveau vers un code prêt à être intégré dans l'architecture finale. Sans raffinement, la modélisation à haut niveau perd de son intérêt et ne fait qu'augmenter le processus de développement des systèmes hétérogènes en terme de temps. Des travaux proposent néanmoins une solution présentée dans le paragraphe suivant.

1.5 La contribution

Comme nous venons de voir à travers ce chapitre, il existe de nombreux travaux dont l'objectif à long terme est d'améliorer le développement des systèmes sur puce. En effet, chacune des méthodologies tend à faciliter, automatiser ou réduire leur temps de développement.

Les langages de description de systèmes hétérogènes haut niveau, l'apparition des RTOS très tôt dans le processus de développement, l'utilisation de niveaux d'abstraction forment des solutions très intéressantes qui, mises en commun permettraient d'obtenir un véritable outil de partitionnement. Remplir cet objectif nécessite également deux conditions qui font partie des contributions de ce travail :

- La possibilité de déplacer un module (décrit en SystemC) du matériel au logiciel ou inversement, du logiciel au matériel en conservant le code de celui-ci et en préservant la communication.
- Une simulation du système hétérogène, grâce à l'intégration d'un RTOS commercial, permet de valider le système partitionné.

Ce mémoire propose donc une méthodologie basée sur la contribution d'un projet visant à développer un outil d'aide au partitionnement des systèmes sur puce s'appuyant sur les deux conditions précédentes. Cette méthodologie fournit un raffinement par niveaux, plus adapté pour la simulation logicielle, tout en permettant

un partitionnement pour chacun d'eux.

Le projet est basé sur une plate-forme de co-design appelée SPACE [6]. Cette plate-forme entièrement développée en SystemC fournit les ressources nécessaires pour le développement d'applications hétérogènes. Cette plate-forme propose actuellement une méthodologie de partitionnement avec deux niveaux de raffinement.

Un des avantages de SPACE est la possibilité d'ordonnancer la partie logicielle décrite en SystemC par un véritable RTOS. Ceci permet la validation de certaines fonctionnalités grâce à la simulation très proche de l'architecture finale et très tôt dans le processus de développement. L'utilisation d'un RTOS est possible en utilisant une interface spécifique entre celui-ci et l'application décrite à haut niveau en SystemC. Un autre avantage est la possibilité de déplacer un module SystemC de la partie matérielle à la partie logicielle ou inversement, sans modifier le code de l'application.

Néanmoins, pour le moment, le nombre de niveaux d'abstraction dans SPACE est limité à deux niveaux : le niveau fonctionnel ne comprenant aucune spécification de l'architecture finale et le niveau cycle où les caractéristiques de la plate-forme comme le type de processeur, le RTOS et le protocole de communication sont pris en compte. L'utilisation de différents niveaux d'abstraction permet donc une décomposition des problèmes. La possibilité de passer d'un niveau à un autre de matière automatique ou ne demandant que très peu d'intervention humaine, est également un gros avantage. Ces facteurs tendent à réduire considérablement le processus d'intégration et réduisent le temps de développement.

Nous allons donc, dans un prochain chapitre, décrire cette méthodologie de raffinement s'appliquant plus particulièrement à la partie logicielle d'un système hétérogène. Nous allons voir comment cette méthodologie vient compléter les ca-

ractéristiques de l'outil d'exploration architectural appelé SPACE qui sera décrit dans le chapitre suivant.

CHAPITRE 2

SYSTEMC ET LA SIMULATION LOGICIEL

Nous l'avons vu précédement : SystemC est une bibliothèque de classes C++ permettant la description d'un système matériel/logiciel à haut niveau, ainsi que sa simulation. Nous allons décrire les différents objets constituant cette librairie. Certain d'entre eux fournissent les caractéristiques de modélisation de composants matériels, et d'autres apportent plusieurs services plus adaptés à la modélisation logicielle. Au cours de ce chapitre nous verrons également les limites concernant la modélisation logicielle de la version SystemC 2.0.1.

2.1 Le langage SystemC

Le langage SystemC est donc composé d'objets de base que l'on peut catégoriser selon la liste suivante :

- Les modules : sc_module.
- Les processus : sc_method, sc_thread, sc_cthread.
- Les ports : sc_port.
- Les signaux : sc_signal.
- Les canaux : sc_channel.
- Les types de données abstraits : sc_int, sc_fx, sc_bit.

Ainsi, les processus sont encapsulés par les modules qui communiquent entre eux par l'intermédiaire de ports reliés entre eux par des canaux. Il existe également d'autres objets pré-définis (sc_fifo, sc_clock, etc.) utilisant un ensemble de ces objets de base.

Seules les fonctions de base seront expliquées au cours de ce chapitre. Pour plus d'information, veuillez vous référer à la documentation fournie avec la bibliothèque SystemC [25], [26].

2.1.1 Les modules

Les modules servent à décomposer le système en différentes entités, plus simple à tester. De plus, cette structure favorise grandement la réutilisation. Pour communiquer avec les autres modules, chacun des modules utilise des ports de communication. Chaque port est connecté à un autre par un canal de communication (signal) permettant ainsi aux éléments du module (variables, processus) de communiquer avec l'extérieur. La figure 2.1 représente les différents objets de base (modules, ports, canaux, interface, processus et module).

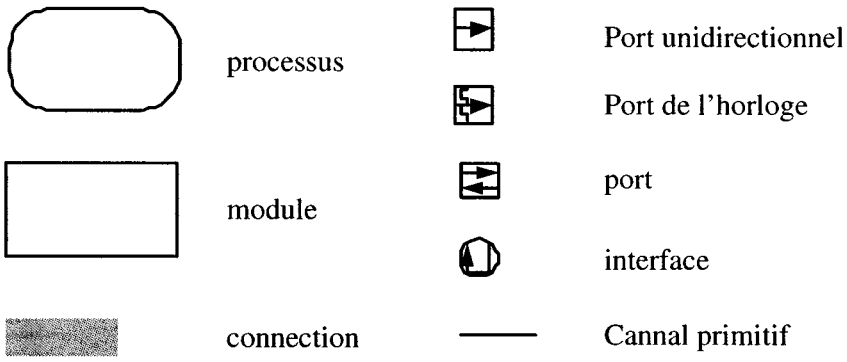

FIGURE 2.1 Objets de base

2.1.2 Les processus

Les processus sont équivalants à des tâches comportant une suite d'instructions à exécuter. Chacun des processus possède une liste de sensibilité permettant d'acti-

TABLEAU 2.1 Les trois types de processus dans SystemC

Processus	Description
SC_METHOD	Processus dont l'exécution ne peut être interrompue par une mise en attente. Ce processus exécute la totalité des instructions qui le compose puis se termine. Il n'est rattaché à aucun processus élémentaire (thread).
SC_THREAD	Processus dont l'exécution est séquentielle et peux donc être suspendue durant son exécution.
SC_CTHREAD	Processus identique au processus SC_THREAD mais sa liste de sensibilité est constitué d'une horloge de sorte que le processus sera activé que lors d'un chargement de front d'horloge.

ver leur exécution. Le simulateur gère l'activation, l'exécution et la mise en attente de chacun de ses processus. D'un point de vu hiérarchique, un processus est toujours contenu à l'intérieur d'un module. De plus, plusieurs processus peuvent être contenus à l'intérieur d'un seul module. Il existe trois types de processus présentés dans le tableau 2.1

Les processus sont équivalents à des tâches qui s'exécutent séquentiellement en contrôlant leur mise en attente par l'ordonnanceur permettant ainsi le transfert d'une tâche à une autre. Elle peuvent décider, par l'intermédiaire d'une instruction (wait) de rendre le contrôle au simulateur afin que celui-ci exécute une autre tâche.

2.1.3 Les ports

Les ports permettent de relier les modules entre eux afin qu'ils puissent communiquer. Il est possible de spécifier les caractéristiques du port c'est-à-dire si le port est un port d'entrée (`sc_in<T>`), un port de sortie(`sc_out<T>`), ou un port d'entré/sortie (`sc_inout<T>`). Les ports sont de type générique (Template) facilitant ainsi leur utilisation avec le type des données qui transigeront par ses ports. Chaque port est connecté à une interface d'un canal ou à un autre port de com-

munication par l'intermédiaire d'un `sc_signal<T>` correspondant à un fils pour la modélisation matérielle.

2.1.4 Les canaux et interfaces

L'interface définit un ensemble de méthodes implémentées dans les canaux qui seront utilisées par les ports pour communiquer. SystemC fourni un certain nombre de canaux (sc_signal, sc_fifo). Le développement d'autres canaux avec leur propre interface est également possible en utilisant la classe abstraite. Ils sont appelés canaux hiérarchiques, ils sont de véritable modules comportant des processus. Ces canaux peuvent être primitifs ou plus complexes, comportant une interface spécifique, définie par l'utilisateur. La figure 2.2 présente un exemple de communication entre deux modules :

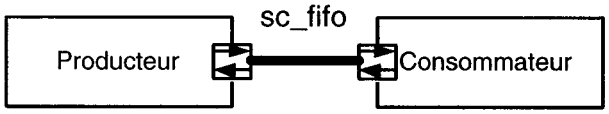

FIGURE 2.2 Exemple d'une implémentation SystemC

2.1.5 Les événements

Les événements permettent d'activer ou de mettre en attente les processus. Ces éléments sont rajoutés dynamiquement au cours de la simulation. Ils sont utilisés pour la synchronisation, par exemple entre les processus.

2.1.6 Les types de données abstraits

SystemC fournit des types de données plus adaptés à la modélisation du matériel (sc_bit, sc_logic, sc_int). Chaque type renferme des méthodes facilitant grandement leur manipulation (concaténation, conversion, opération arithmétique et logique).

2.1.7 L'horloge

En matériel, l'horloge est très importante puisqu'elle permet de synchroniser les différents composants d'un système, tant matériel que logiciel. De type sc_clock, l'horloge est entièrement configurable (période, et caractéristique du signal).

2.2 L'ordonnanceur

L'ordonnanceur détermine l'ordre d'exécution des processus. Cet ordre est déterminé en fonction de la liste de sensibilité des ports et du déclenchement des événements intervenant dans le système. Le simulateur de SystemC est très similaire au simulateur VHDL puisqu'il est basé sur le delta-cycle. Le delta-cycle est la portion de temps au cours de laquelle le système détermine l'ordre d'exécution des processus dans chaque période de l'horloge. Chaque delta-cycle est composé d'une phase d'évaluation pour déterminer les processus prêts à être exécutés et d'une phase de mise à jour destinée à mettre les sorties à jour. Il est possible que plusieurs itérations de delta-cycle interviennent au cours d'un cycle d'horloge. Le simulateur peut également mettre un processus en attente pour un temps donné à l'aide de l'instruction wait(nb) où "nb" spécifie la durée d'attente en nombre de cycles ou en temps. Le schéma 2.3 présente le parcours effectué par le simulateur.

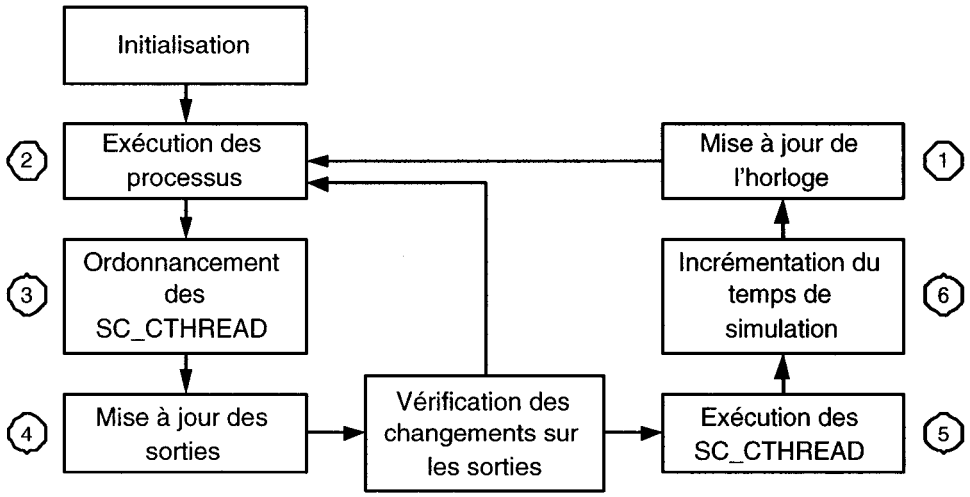

FIGURE 2.3 Simulateur SystemC

1. Incrémentation des horloges.
2. Exécution des SC_METHOD/SC_THREAD qui ont un changement à leur entrée.
3. Mise à jour des sorties des processus de type SC_CTHREAD. Leur exécution se fera à l'étape 5.
4. Si des changements surviennent sur une ou plusieurs sorties, recommencer à partir de l'étape 2.
5. Exécution des SC_CTHREAD. Les sorties seront propagées à l'étape 3 du prochain coup d'horloge.
6. Incrémentation du temps de simulation et redémarrage à l'étape 1.

Du point de vue de l'implémentation, étant donné que les threads (processus) sont des unités d'exécution indépendantes, le code du changement de contexte (appels systèmes utilisés par l'ordonnanceur afin de migrer d'une tâche à une autre) est dépendant de l'architecture utilisée. C'est pourquoi SystemC utilise des

implémentations de threads différentes suivant la plate-forme de développement utilisée.

- Environnement Windows : l'implémentation des threads utilisés est appelé *Fiber* (se référer à la documentation MSDN). Un *Fiber* est une unité d'exécution qui est ordonnancé manuellement par l'application. Il s'exécute dans le même contexte que le processus (l'application) qui les ordonnance, ici SystemC (le mécanisme est comparable à celui des Thread POSIX).
- Environnement Linux : l'implémentation des threads est basée sur une librairie appelée QuickThread développée par l'Université de Washington en 1993 [20]. Cette librairie regroupe un ensemble d'outils permettant de créer et de gérer (passage d'un contexte à un autre) des unités d'exécution (thread).

Nous venons de voir que l'ordonnanceur SystemC utilise des unités d'exécution pour gérer les différents processus, et qu'il s'apparente grandement aux simulateurs de VHDL, donc aux simulateurs matériels. Dans ce qui suit, nous allons voir ce qu'il en est au niveau de la simulation logicielle.

2.3 Les limitations logiciels de SystemC

A travers cette section nous allons voir que le simulateur de SystemC renferme quelques lacunes quand à la simulation de la partie logicielle d'un système. Cependant. Certaines améliorations sont prévues pour la version 3.0 de cette bibliothèque. Dans ce travail, nous parlerons essentiellement de la version courante disponible, la version 2.1. L'ordonnancement matériel est différent de celui du logiciel. En effet, en matériel, tous les processus s'exécutent dans la même période de temps. C'est ce que fait le simulateur SystemC. A chaque intervalle de temps, les processus prêts à être exécutés sont exécutés les uns à la suite des autres, donnant ainsi l'impression qu'ils s'exécutent en même temps, comme dans la réalité.

Au niveau logiciel, l'ordonnancement est différent. Le processeur ne peut exécuter tous les processus dans une seule et même période de temps. Chacun des processus est donc exécuté pendant plusieurs périodes de temps, selon un ordre déterminé par exemple, par la priorité. Le simulateur de SystemC ne propose pas pour le moment, cette nuance. Toutefois, des travaux ont été réalisés dans cette voie, comme nous l'avons vu dans le chapitre de la revue de littérature. La deuxième caractéristique vient du fait que le simulateur SystemC n'est pas préemptif. Un système est dit préemptif lorsque son ordonnanceur répartit selon, par exemple, les critères de priorité, l'ordre d'exécution entre les différentes tâches qui en font la demande (à chaque ré-ordonnancement lors d'une interruption de l'horloge ou lors d'un ré-ordonnancement volontaire). Ici, concernant notre simulateur, le changement de contexte n'a lieu seulement que lorsque la tâche en cours d'exécution termine son exécution (dans le cas d'un processus SC_METHOD), ou se met en attente pour une période de temps, d'un nombre de cycle, d'un ou plusieurs événements en relation ou non avec les communications, ou d'une combinaison des deux (dans le cas des processus SC_THREAD et SC_CTHREAD).

2.4 Conclusion

L'idée de modifier la librairie SystemC ne constitue pas une bonne solution. Une version 3 est actuellement en cours de développement mais l'approche préconisée n'est pas connue. Par conséquent, la librairie pourrait ne plus être compatible avec la version officielle qui est sur le point de devenir un standard. Une divergence pourrait donc nuire à la philosophie SystemC. Notons de plus que l'organisation responsable du développement du maintient et de l'évolution de SystemC, l'OSCI, est constituée d'un groupe de compagnies et d'universités dont le rôle est d'influencer le développement de SystemC en participant également au développement et en

votant les différentes solutions techniques à adopter. Malheureusement, le Groupe de Recherche en Microélectronique (GRM) de l'école Polytechnique de Montréal ne fait pas parti de l'OSCI. Comment donc rendre possible une modélisation plus fidèle de la partie logicielle d'un système à l'aide de SystemC ? SystemC n'est pas suffisamment adapté pour la simulation de l'application matérielle et logicielle. Plutôt que de vouloir intégrer la simulation logicielle, la solution suivante est envisagée : l'ajout d'un véritable OS commercial destiné à cette simulation logicielle. C'est pourquoi une architecture matérielle équipée d'un simulateur de processeur permettant ainsi d'exécuter un véritable RTOS a été développée. Celle-ci fournit l'ordonnancement idéal pour les processus logiciels. C'est le travail que nous allons décrire dans le prochain chapitre.

CHAPITRE 3

SPACE : UNE ARCHITECTURE D'AIDE AU PARTITIONNEMENT DES SYSTÈMES SUR PUCES

SPACE (SystemC Partitioning of Architectures for Co-design of Embedded system) est une plate-forme SystemC destinée à l'exploration architecturale. Ce processus tend à déterminer (en essayant plusieurs configurations du partitionnement de l'application) la répartition matérielle/logicielle qui satisfait les spécifications. Les caractéristiques de l'outil présenté ici résident dans le fait de permettre le partitionnement d'une application décrite en SystemC. Le partitionnement consiste en une division d'un système en deux parties : logicielle et matérielle. Ce partitionnement n'est pas fixe et peut à tout moment être reconsidéré. C'est à dire qu'un module SystemC peut être facilement déplacé du matériel au logiciel ou inversement. De plus, il intègre un véritable RTOS commercial permettant l'ordonnancement de la partie logicielle décrite en SystemC. Cette fonction est réalisée par l'intermédiaire d'une API SystemC faisant le lien entre les mots clés SystemC et les appels de fonctions spécifiques au RTOS utilisé.

3.1 La philosophie

L'aspect novateur réside dans le fait que les parties logicielles et matérielles sont décrites dans un seul et même langage et simulées par un ordonnancement matériel (celui de SystemC) pour la partie matérielle et un ordonnancement logiciel (celui du RTOS) pour la partie logicielle.

L'application doit tout d'abord être spécifiée à haut niveau, dans un seul et même

langage : SystemC. La conception de chacun des modules doit répondre à un style de programmation plutôt comportementale en respectant certains critères (par exemple : l'utilisation du même protocole de communication du coté logiciel et matériel) de manière à pouvoir être éventuellement déplacé facilement de la partie matérielle vers la partie logicielle ou inversement. Une fois la fonctionnalité du système validée, le partitionnement est la prochaine étape. Chacun des modules est alors connecté soit à la partie matérielle ou logicielle, formant ainsi une configuration. Plusieurs configurations vont pouvoir être simulées de manière à déterminer le partitionnement optimal satisfaisant les spécifications. Ceci est possible en déplaçant un module dans la partie logicielle ou matérielle. Cette étape est simple puisque chacune des parties utilise une interface de communication identique. Le code de l'application n'a pas besoin d'être modifié. Une simple recompilation permet d'obtenir les nouveaux résultats de la simulation. Ce processus itératif peut donc recommencer jusqu'à l'obtention d'un partitionnement optimal.

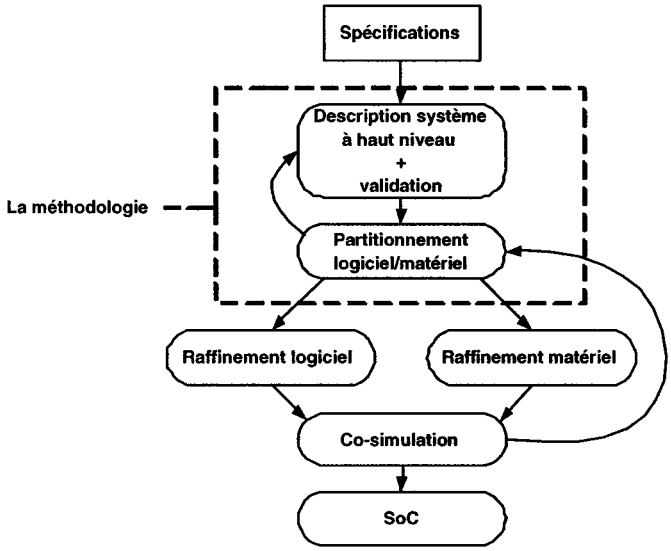

FIGURE 3.1 Situation de la méthodologie dans le cycle de développement

Nous allons maintenant décrire les principales caractéristiques de cette architecture. Ceci va nous permettre de mieux comprendre l'implémentation de la méthodologie que nous verrons au chapitre suivant.

3.2 Description de la plate-forme

SPACE propose donc une architecture matérielle décrite en SystemC. Cette partie matérielle est composée de modules et de périphériques (horloge, mémoire, gestionnaire d'interruption), d'un gestionnaire de communication et d'un émulateur de processeur (ISS). Cette configuration de base fournit les ressources nécessaires au fonctionnement d'une application hétérogène. Elle accueille, en effet, des composants matériels spécifiques à l'application qui, connectés au gestionnaire, pourront communiquer avec le reste de la plate-forme. Ce travail a été réalisé par Olivier Benny [3], participant également au projet. La présence de l'émulateur de processeur (ISS) permet l'exécution d'un système d'exploitation temps réel. Cette implantation de l'ISS a été réalisée par Jérôme Chevalier [5]. La contribution de ce mémoire se situe au niveau de l'intégration du RTOS ainsi que le développement d'une API SystemC permettant d'interpréter le code de l'application écrit en SystemC afin que le RTOS soit en mesure d'interpréter et d'ordonnancer les tâches (processus) de l'application logicielle. Les modules placés dans la partie logicielle sont compilés avec l'API SystemC et le RTOS. Le code binaire ainsi obtenu est placé dans le module mémoire afin d'être exécuté par le processeur (l'ISS) de la plate-forme. Ainsi, la gestion de la préemption entre les différents processus logiciels, est rendue possible. Nous verrons les détails de la partie logicielle dans un prochain chapitre.

Au niveau matériel, un protocole de communication a également été implémenté. Celui-ci permet d'assurer la communication entre les modules, qu'ils soient logiciels

ou matériels, tout en garantissant leur déplacement d'une partie à l'autre (du logiciel vers le matériel ou inversement). Ceci, dans l'objectif de garantir ainsi l'exploration architecturale. Les modules de l'application placés dans la partie matérielle, sont connectés au gestionnaire de communication (équivalent à un bus avec un arbitre permettant d'aiguiller les messages qui transitent sur celui-ci) à travers des adaptateurs. Un module très important appelé décodeur, joue le rôle de connecteur faisant le lien entre la partie logicielle et la partie matérielle. C'est par lui que transigent les communications. La figure suivante présente la plate-forme SPACE composée de ses modules et périphériques.

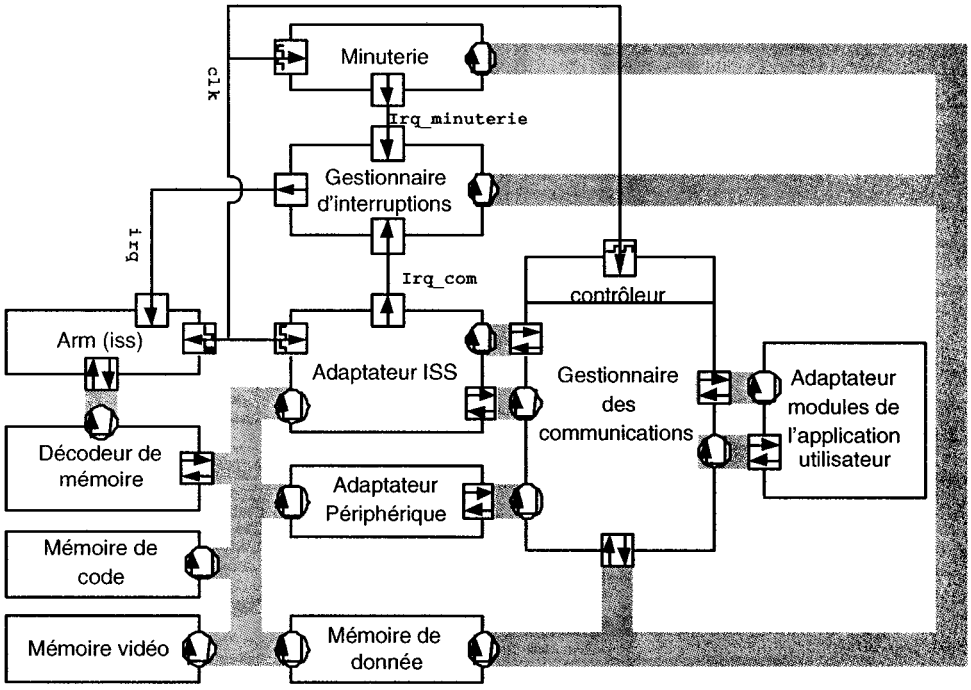

FIGURE 3.2 Architecture de SPACE présentant ses modules et périphériques

- Un module peut initier une transaction à destination d'un autre module ou d'un périphérique sur le canal de communication. Il est considéré comme étant un maître. Il possède un identifiant unique attribué par le développeur. Cet

identifiant correspond à la priorité du module lorsque celui-ci se trouve dans la partie logicielle.

- Un périphérique est un esclave qui répond aux requêtes des modules. Il a une plage d'adresses définie lors de sa configuration, dépendamment de ses spécifications.

Dans ce qui suit, nous allons présenter les différents bloques constituant la plateforme SPACE.

3.2.1 La minuterie

La minuterie est un périphérique composé d'un compteur qui envoie une interruption de manière périodique. Cette période est paramétrable à l'aide des fonctions d'initialisation lors du démarrage du système. Une interruption est ensuite générée à chaque fois que la valeur du compteur atteint la valeur initiale.

3.2.2 Le gestionnaire d'interruptions

Le gestionnaire d'interruptions permet d'augmenter le nombre d'entrées d'interruption que peux gérer le processeur. Le processeur utilisé ici (le processeur ARM) ne possède que 2 entrées pour les interruptions : nIRQ et nFIQ. Le gestionnaire vient augmenter le nombre d'interruptions grâce à ses multiples entrés. Sa sortie est directement reliée à l'entrée nIRQ du processeur. Actuellement l'interruption de la minuterie et celle de l'adaptateur (chargé des communications) du processeur sont reliées au gestionnaire d'interruption. La figure 3.3 illustre les explications précédentes.

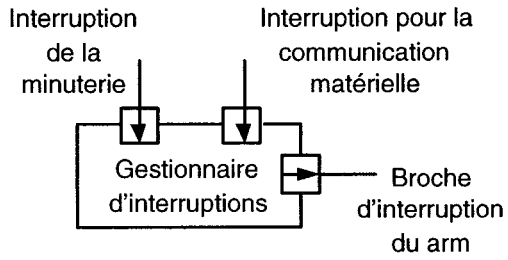

FIGURE 3.3 Gestionnaire d'interruptions

3.2.3 Le processeur

La plate-forme est composée d'un processeur qui est en fait un simulateur de processeur (Instruction Set Simulator) permettant d'émuler le comportement d'un véritable processeur. Ceci permet de simuler conjointement la partie logiciel et matérielle de la plate-forme.

Cet ISS exécute un code binaire (des instructions) qui a préalablement été compilé pour ce type de processeur. L'ISS, basé sur l'architecture ARM va donc lire, décoder et exécuter une à une chaque instruction constituant le code binaire de l'application. Le travail, effectué par Jérôme Chevalier a consisté à adapter l'ISS ARM présent dans l'outil de déverminage de GNU [14] nommé GDB [12] afin qu'il puisse fonctionner sur la plate-forme. Pour ce faire, l'ISS a tout d'abord été encapsulé dans un module SystemC puis ensuite, plusieurs fonctionnalités ont été rajoutées :

- La synchronisation avec l'horloge du système. De cette manière l'ISS exécute une instruction à chaque front d'horloge de la plate-forme.
- La prise en compte des interruptions. Un mécanisme a été rajouté permettant à l'ISS d'interrompre l'exécution en cours et de changer son pointeur de programme afin d'exécuter du code (à une adresse différente) correspondant à la routine d'interruption.

- Le chargement des instructions. Le module ISS utilise la mémoire de code de la plate-forme par l'intermédiaire d'un port de communication dédié.

L'ISS est synchronisé avec le reste de la plate-forme à l'aide d'une mise en attente sur l'horloge du système. La communication est assurée par l'intermédiaire des ports en utilisant les fonctions de lecture et d'écriture. Pour des raisons de synchronisation la fréquence du processeur est identique au reste de la plate-forme. Ce choix permet de simplifier le développement.

3.2.4 Les mémoires

Il existe deux mémoires sur la plate-forme :

- La mémoire de code comprenant la partie logicielle codée pour être exécutée par le processeur.
- La mémoire de donnée qui, comme son nom l'indique, comporte toutes les variables nécessaires au fonctionnement de la plate-forme et de l'application.

3.2.5 Le décodeur mémoire

Le décodeur mémoire permet de répondre aux requêtes de lecture ou d'écriture, adressées par le processeur. Le décodeur interprète l'adresse en provenance du processeur, puis active le module correspondant (mémoire de code, mémoire de données, adaptateur pour les périphériques ou adaptateur pour les modules matériels de l'application (adaptateur ISS)). Il écrit l'information dans le cas d'une écriture ou renvoie la valeur contenue à l'adresse correspondante si c'est une lecture.

3.2.6 Le gestionnaire des communications

Le gestionnaire des communications est le module sur lequel les autres modules et les périphériques vont être connectés par l'intermédiaire des adaptateurs (adapter) afin de pouvoir communiquer entre eux. Ce gestionnaire de communication joue le rôle d'un bus qui, contrôlé par un arbitre, permet d'envoyer le message au bon destinataire.

3.2.7 Les adaptateurs des modules de l'application utilisateur

Ces adaptateurs sont l'interface entre les modules matériels de l'application et le gestionnaire des communications. Ses modules répondent aux requêtes de lecture et d'écriture provenant des modules de l'application. Ils contiennent les messages envoyés par les autres modules aussi longtemps que le module qui lui est connecté envoie une requête de lecture. Ceci permet d'abstraire le canal de communication de la modélisation de l'application. Ce canal de communication pourra par la suite être remplacé par un autre modèle sans modifier le code de l'application.

3.3 Conclusion

La plate-forme est donc un ensemble de modules interconnectés entre eux réalisant une véritable structure d'accueil (mémoires, processeur, périphériques) permettant l'exécution d'une application SystemC composée d'une partie matérielle et d'une autre logicielle. Des outils sont également disponibles pour aider au développement d'applications hétérogènes. Un outil de déverminage a été développé pour la partie logicielle. Celui-ci accompagné d'un dévermineur pour la partie matérielle permet de suivre l'exécution de l'application pas à pas, tant du coté matériel que logiciel sur

la même plate-forme. Nous allons nous attarder plus en détail sur la méthodologie dans le prochain chapitre. Nous verrons que celle-ci propose trois niveaux d'abstraction permettant le partitionnement logiciel/matériel d'une application.

CHAPITRE 4

MÉTHODOLOGIE

4.1 Introduction

L'utilisation de langages de description à haut niveau concernant la partie logicielle reste encore difficile dû à la complexité lors de la synthèse. En effet, le code de l'application doit, la plupart du temps, être modifié afin d'intégrer l'architecture finale. Toute modification de code lors de la phase d'intégration peut être source d'erreur et doit être évitée. La méthodologie proposée dans ce mémoire regroupe trois niveaux permettant un raffinement logiciel utilisant SystemC 2.01 à partir d'une description à très haut niveau jusqu'à la synthèse dans un système sur puce. Un premier niveau, appelé L1 (Level1) permet la spécification de l'application ainsi que sa validation fonctionnelle à l'aide du simulateur SystemC. Au deuxième niveau, appelé L2 (Level2), l'application est partitionnée en deux parties : les modules logiciels et les modules matériels. La partie matérielle est exécutée avec le simulateur de SystemC, tandis que la partie logicielle est ordonnancée par un RTOS exécuté comme étant un processus encapsulé dans une API SystemC. Enfin, au troisième niveau, appelé L3 (Level3), chacune des partitions est connectée sur la plate-forme SPACE, incluant le même RTOS s'exécutant sur un émulateur de processeur ARM (ISS) ordonnancé par le simulateur de SystemC. La contribution de la méthodologie réside dans le fait de permettre un partitionnement à chacun des trois niveaux d'abstraction. Cette caractéristique permet l'exploration architecturale ainsi que le raffinement logiciel dans le but de faciliter la phase délicate d'intégration.

4.2 Présentation des trois niveaux

Comme mentionné précédemment, la méthodologie présente trois niveaux d'abstraction. Un premier niveau (L1) permet la description du système et sa validation fonctionnelle. L'application est ensuite partitionnée au niveau L2 où elle est simulée puis, au niveau L3, l'application est connectée sur la plate-forme SPACE. La figure 4.1 présente ses trois niveaux. Chacun des niveaux est composé de trois couches logicielles :

1. L'application développée en SystemC : L'application est constituée en modules comportant des processus et utilisant un port de communication spécifique, propre à la plate-forme que nous décrirons dans le chapitre suivant.
2. L'API SystemC : Cette couche joue plusieurs rôles : a)l'initialisation et la communication entre le RTOS et l'application SystemC et b)la gestion des communications en assurant le transfert de messages d'un module à un autre (matériel ou logiciel).
3. Le RTOS : Cette couche contient l'ordonnanceur des tâches logicielles ainsi que l'HAL [34](Hardware Abstraction Layer) dépendamment du niveau d'abstraction. En effet, celle dernière est différente pour le niveau L2 et L3. Au niveau L2, elle permet au RTOS d'être exécuté comme un processus Linux tandis que, au niveau L3 celle-ci permet à ce dernier d'être exécuté sur la plate-forme SPACE utilisant l'ISS ARM et les différents périphériques.

Le RTOS utilisé pour valider l'approche présentée dans ce travail est MicroC/OS-II [21]. Celui-ci offre tous les avantages d'un système d'exploitation temps réel : un ordonnancement préemptif des tâches, basé sur les priorités, et gérant également les interruptions. MicroC/OS-II a été choisi en raison de sa faible complexité, de la disponibilité de son code source et parce qu'il est déjà porté sur une vaste gamme de processeurs notamment le processeur ARM émulé sur la plate-forme SPACE.

La figure 4.1 présente les trois niveaux d'abstraction :

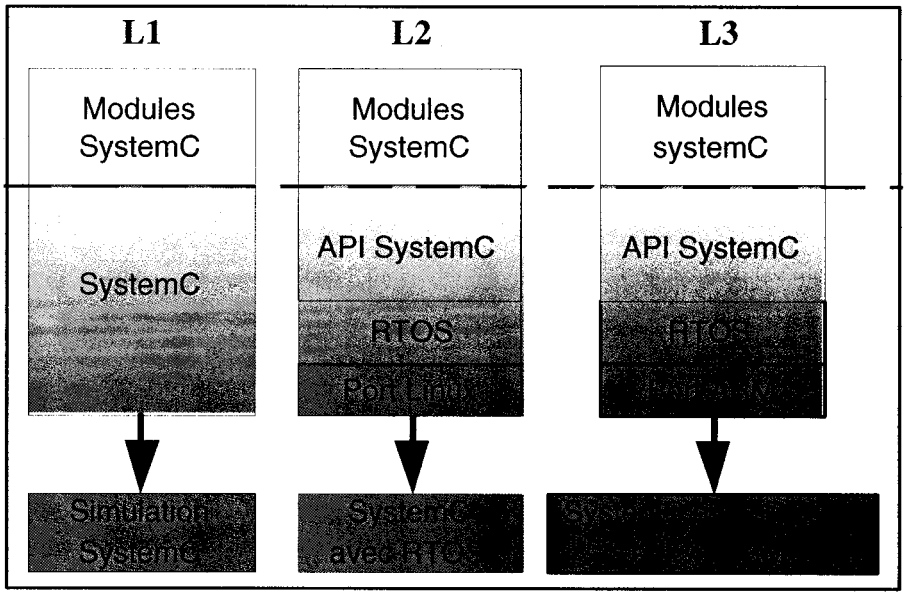

FIGURE 4.1 Les niveaux d'abstraction

Comme nous l'avons vu précédemment, l'API SystemC possède plusieurs fonctions. Elle est tout d'abord responsable de la phase d'initialisation pendant laquelle tout l'environnement logiciel est construit (la liste des différents processus logiciels). Ensuite, le RTOS est initialisé, chacun des processus devient alors une tâche pour le RTOS. L'ordonnanceur est ensuite activé. Le deuxième rôle de cette API SystemC est de jouer l'interprète entre le RTOS et l'application écrite en SystemC. En effet, sa fonction consiste à convertir les appels de fonctions SystemC utilisés par l'application en appels de fonctions propres au RTOS. Les travaux décrits dans l'article [17] utilisent une interface similaire, incluant un RTOS. La différence ici est que l'API permet de déplacer les modules en logiciel ou en matériel. Ceci est possible grâce à l'utilisation d'un port de communication spécifique , dont l'interface est identique pour le matériel et le logiciel. La gestion des communications entre

les modules constitue le troisième grand rôle de l'API SystemC. L'objectif est l'acheminement des messages d'un module à l'autre, qu'il soit logiciel ou matériel.

Nous verrons plus en détail les caractéristiques de cette API SystemC dans un chapitre suivant. Nous allons pour le moment expliquer plus en détail les différents niveaux d'abstraction.

4.3 Niveau L1

Dans un premier temps, l'application est décrite entièrement en SystemC, non partitionnée (figure 4.2). Comme mentionné à la section 4.2, la description des modules constituant l'application doit suivre certaines spécifications. Celles-ci sont regroupées dans la liste suivante :

- Un module ne peut contenir qu'un seul processus.
- Ce processus doit être soit de type SC_THREAD ou SC_CTHREAD. Le type SC_METHOD n'est pas implémenté pour le moment.
- l'attribution d'une étiquette unique pour chacun des module. Cette étiquette correspond également à la priorité du module lors d'un ordonnancement logiciel.
- Un seul port de communication, spécifique à la plate-forme SPACE, relie le module au reste du système afin de garantir le déplacement de ce module dans la partie logicielle ou matérielle.
- Utilisation d'un niveau de représentation comportemental pour la description des différents modules. Leur implémentation interne est indépendante de l'architecture finale permettant plusieurs choix de partitionnement possible.

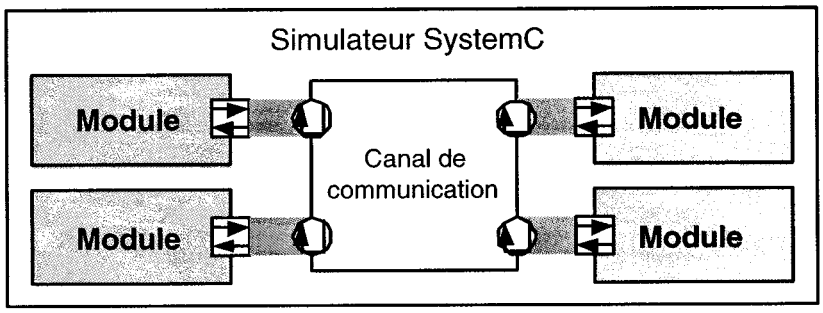

FIGURE 4.2 Niveau d'abstraction L1

Afin d'assurer la communication indépendamment du partitionnement, chaque module possède un seul et unique identifiant attribué par le concepteur. Notons également que cet identifiant correspondra à la priorité du module si celui-ci se trouve dans la partie logicielle et donc ordonnancé par le RTOS. La communication est similaire au transfert de paquets sur un réseau. Chaque message est encapsulé et comprend une entête contenant l'identification de l'envoyeur, celui du destinataire, puis la taille du message envoyé. Une fois les fonctionnalités validées par le simulateur de SystemC, l'application est partitionnée au niveau L2.

4.4 Niveau L2

Ce deuxième niveau concerne le partitionnement de l'application. Les modules de l'application peuvent être placés soit dans la partie logicielle ou dans la partie matérielle du système. Ce premier choix est, pour le moment, déterminé selon l'expérience ou le bon sens du concepteur. Le choix du partitionnement n'est pas fixe et peut à tout moment être reconsidéré. Les modules matériels sont exécutés par le simulateur SystemC alors que les modules logiciels sont exécutés dans un processus différent. Ils sont, en effet, ordonnancés par le RTOS s'exécutant comme un processus Linux. Le système d'exploitation (Linux) exécute le RTOS (MicroC/OS-II) fournissant ainsi un ordonnancement préemptif, basé sur la priorité, même s'il

n'est pas exécuté sur la plate-forme finale, avec le processeur final. Comme expliqué précédemment, l'ordonnancement des modules SystemC est possible grâce à l'API SystemC située entre l'application et le RTOS. Ce niveau fournit des résultats sur la fonctionnalité du système indépendamment de l'architecture finale et donc sans aucune information au niveau cycle. A ce niveau, l'attribution des priorités peut être remise en cause afin d'obtenir le comportement désiré si celui-ci ne satisfait pas les spécifications. La simulation à ce niveau doit d'être plus rapide que le niveau inférieur (L3). La figure 4.3 présente la configuration du système ainsi qu'une application partitionnée avec d'un côté les modules connectés à la partie logicielle et de l'autre, les modules connectés à la partie matérielle.

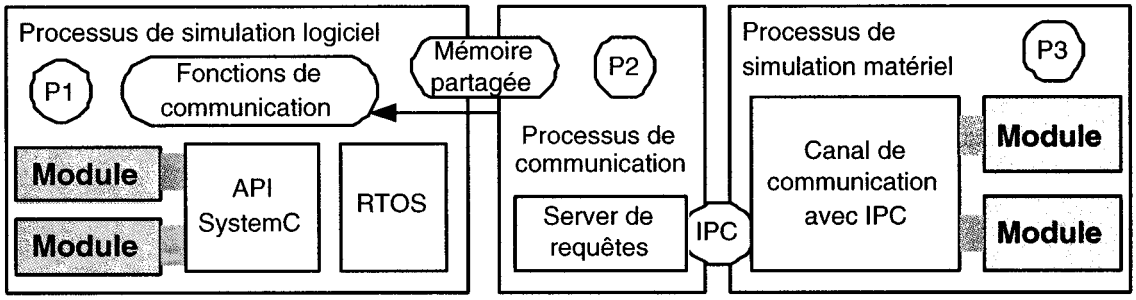

FIGURE 4.3 Niveau d'abstraction L2

La simulation logicielle est constituée de trois processus : P1, P2 et P3 (figure 4.3). Le processus P1 est constitué de l'API SystemC du RTOS ainsi que des modules logiciels de l'application. Le RTOS a donc été porté afin de s'exécuter sur Linux en utilisant les ressources disponibles. En effet, la minuterie et les signaux POSIX [4] permettent d'émuler les interruptions (section 5.3.1). Le processus P3 est chargé de la simulation des modules matériels. Enfin, le processus P2 est responsable de la communication entre la partie matérielle (P3) et la partie logicielle (P1). La simulation est synchronisée par les messages qui transitent entre ses deux processus

(P1 et P3). Le processus matériel envoie des requêtes au processus de communication (P2) en utilisant un mécanisme de type IPC (fourni par Linux) basé sur les sockets. Les requêtes envoyées par le processus P3 (processus matériel) peuvent être de différents types :

- La liste des étiquettes des modules logiciels. Cette requête est utilisée pendant la phase d'initialisation.
- Une lecture ou écriture à destination d'un module appartenant à la liste obtenue par la requête précédente.
- La signalisation de la fin de la simulation matérielle. Cette requête permet d'interrompre le processus simulant la partie logicielle de l'application.

Si la requête est une opération de lecture ou d'écriture, le processus de communication reçoit la requête puis l'écrit dans un pipe POSIX. Ensuite, un signal est envoyé au processus logiciel afin de l'interrompre et d'exécuter la fonction de récupération de la requête. Celle-ci est ensuite décodée, et envoyée au gestionnaire de communication de l'API SystemC. Le message est alors transmis dans la boîte de réception du destinataire (dans le cas d'une écriture) ou lu, dans le cas d'une lecture. Puis, la réponse est renvoyée au processus de communication pour être encapsulée et envoyée au processus matériel en attente de la réponse.

4.5 Niveau L3

Le niveau L3 est entièrement basé sur la plateforme SPACE. Les modules testés dans les niveaux un et deux (L1 et L2) sont réutilisés, sans modification de leur code, et placés sur la plate-forme d'exploration architecturale. La configuration du système (son partitionnement) peut également être remise en cause et modifiée à ce niveau, dépendamment des résultats de simulation obtenus au niveau L2. Ce niveau d'abstraction se rapproche de l'architecture finale puisqu'il permet l'exécution

de l'application avec les spécifications de l'architecture finale (espace d'adressage, processeur). Actuellement, seul le processeur ARM est supporté par SPACE. Les modules sont donc compilés afin d'être exécutés par l'émulateur de processeur de type ARM (l'ISS). La différence avec le niveau L2 réside dans la communication entre les modules matériels et logiciels ainsi que le port du RTOS (la couche HAL) qui permet à celui-ci de pouvoir s'exécuter sur le processeur de type ARM et d'utiliser les différents périphériques présents sur la plate-forme SPACE (la minuterie, le contrôleur d'interruptions ainsi que la mémoire et les registres). En effet, le RTOS n'est plus encapsulé dans un processus comme dans le niveau L2, il joue le rôle du système d'exploitation principal.

Le gestionnaire des communications utilise des fonctions spécifiques afin d'envoyer ou de recevoir des messages de la partie matérielle. A travers le gestionnaire, un module peut accéder aux divers périphériques ou envoyer un message à un autre module qu'il soit matériel ou logiciel. Lorsqu'un module matériel veut envoyer un message à la partie logicielle, une interruption est générée par le gestionnaire des interruptions à destination de l'ISS. Le RTOS interrompt son ordonnancement et une routine permet alors de récupérer le message de la partie matérielle et de l'envoyer au gestionnaire des communications logicielles qui sera chargé d'acheminer le message au destinataire. La figure 4.4 montre comment les modules logiciels et matériels peuvent communiquer.

Un périphérique spécifique appelé Adaptateur ISS (figure 3.2) est utilisé pour faire le liens entre la mémoire et le canal de communication matériel. Rappelons que la plate-forme SPACE regroupe les périphériques nécessaires au fonctionnement des RTOS (minuterie et gestionnaire d'interruptions, espace d'adressage, mémoire) et

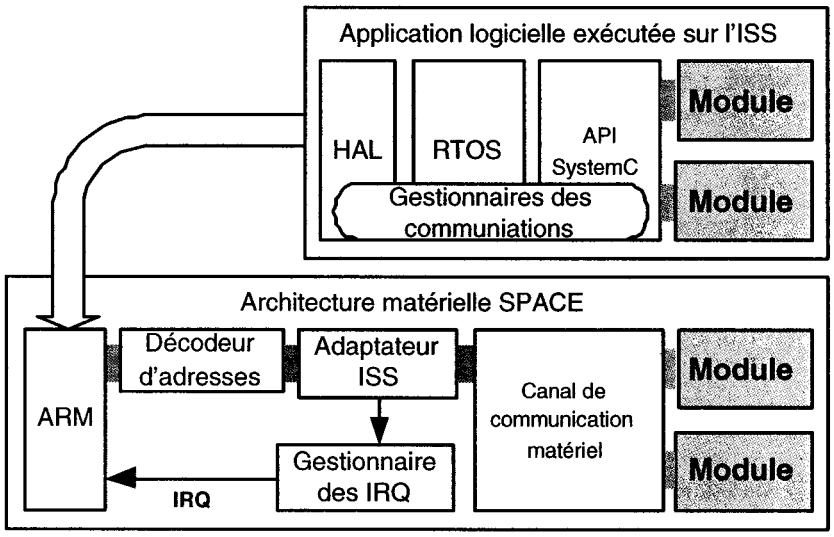

FIGURE 4.4 Niveau d'abstraction L3

il est également possible d'ajouter d'autres fonctionnalités permettant ainsi d'avoir les mêmes caractéristiques que la plate-forme finale.

Afin de pouvoir suivre le déroulement de l'exécution, et également de procéder à des corrections, il est possible d'utiliser une dévermineur spécialement conçu pour remplir cette tâche. Cet outil est basé sur le dévermineur Insight [18], une interface graphique de GDB [12] (le dévermineur de GNU [14]). L'ISS de SPACE a donc été modifié afin de permettre une communication avec Insight. Celui-ci peut donc fournir des informations sur le déroulement de l'exécution de la partie logicielle. Tout ceci permet de fournir un environnement de co-simulation puisqu'il est possible de suivre l'exécution de l'application aussi bien du côté logiciel que du côté matériel avec deux dévermineurs différents. Un premier destiné au suivi de la partie matérielle exécutée par le simulateur SystemC, tandis qu'un deuxième en charge de la partie logicielle, ordonnancée par le RTOS exécuté sur l'ISS.

4.6 Récapitulatif

Dans ce chapitre, nous avons présenté la méthodologie de raffinement de la partie logicielle utilisant la plate-forme SPACE. Cette méthodologie, basée sur trois niveaux, permet une abstraction progressive des spécifications et contraintes de l'architecture finale. Nous avons montré que le partitionnement de l'application, pouvait être remis en cause à chaque niveau si les résultats ne satisfaisaient pas les spécifications. Dans le chapitre suivant, nous allons décrire la partie logicielle qui permet à un RTOS commercial d'ordonnancer des modules SystemC. Puis, nous nous attarderons sur le gestionnaire des communications de cette API SystemC.

CHAPITRE 5

INTERFACE DE PROGRAMMATION SYSTEMC

Ce chapitre décrit l'environnement logiciel nécessaire aux différents niveaux d'abstraction présentés dans la méthodologie au chapitre précédent.

5.1 Objectifs

Comme nous l'avons vu, la méthodologie propose trois niveaux de raffinement pour la partie logicielle : une description haut niveau du système (L1), un premier partitionnement avec des premiers résultats de simulation (L2) et une simulation sur une plate-forme d'exploration architecturale (SPACE) (L3).

Afin de garantir toutes ses caractéristiques, ainsi que le passage d'un niveau à l'autre le plus simple possible, un support logiciel doit être créé. Les fonctions du support logiciel sont les suivantes :

- Fournir un support indispensable permettant la description haut niveau avec SystemC. Pour ce faire, une API va permettre d'interpréter l'application décrite en SystemC, un langage de description à haut niveau.
- Garantir le partitionnement de l'application. C'est à dire être capable de garantir la communication entre les modules, peut importe leur nature logicielle ou matérielle.
- Permettre une simulation la plus fidèle possible à l'environnement final dans lequel il sera intégré. L'accent a été mis sur l'ordonnancement logiciel. Pour ce faire, toutes les fonctions nécessaires pour accueillir un véritable RTOS commer-

TABLEAU 5.1 Lacunes de SystemC face à l'ordonnancement logiciel

Services offerts par un environnement logiciel	SystemC 2.0	RTOS
Gestion du temps par le microprocesseur ou le micro-contrôleur en assurant l'exécution de la tâche la plus importante	Non	Oui
Gestion du multi-tâches	Oui	Oui
Décomposer l'application en plusieurs tâches comportant chacune une tâche spécifique	Oui	Oui
Fournir des services spécifiques (mise en attente, gestion des sémaphores, communication, synchronisation)	Partiellement	Oui
Prise en compte des événements en temps réel	Non	Oui
Attribution de priorités dans les tâches à exécuter	Non	Oui

cial qui, avec peu d'effort, peut être remplacé par celui qui sera utilisé sur la cible.

Nous allons donc détailler ses différentes fonctionnalités et voir ce que cela implique au niveau de la conception.

Comme nous l'avons vu précédemment, l'ordonnancement SystemC est plus approprié pour l'ordonnancement matériel puisqu'il possède beaucoup de similitudes avec les simulateur de type VHDL. Afin de permettre une simulation du comportement logiciel, nous proposons donc d'intégrer un ordonnancement logiciel. Pour ce faire, plutôt que de développer et intégrer un modèle de RTOS dans SystemC, notre choix s'est porté sur l'intégration d'un véritable RTOS facilement interchangeable. Le tableau suivant présente des fonctionnalités non encore supportées dans la version actuelle de SystemC 2.0 concernant la simulation des tâches logicielles.

À travers le tableau 5.1, il parait évident que l'utilisation d'un RTOS dans le processus de modélisation viendrait combler ce que ce langage de description haut niveau ne fournit pas actuellement : un ordonnancement logiciel fidèle à celui fourni

par un véritable RTOS. De plus, l'utilisation du véritable RTOS très tôt dans le processus de développement est un atout considérable pour l'intégration finale.

Rappelons que cette configuration concerne les niveaux L2 et L3 de la méthodologie. Comme nous l'avons décrit précédemment (section 4.2) la partie logicielle est constituée de plusieurs couches logicielles : l'application écrite en SystemC, le RTOS et l'API SystemC. La figure 5.1 montre la structure de la partie logicielle et présente les différentes couches. Le RTOS commercial porté pour l'architecture (dépendamment du niveau d'abstarction L2 ou L3) est exécuté par le processeur. L'API SystemC sert de traducteur entre ce RTOS et les modules de l'application écrits en SystemC. Cette API renferme les mots clés de la bibliothèque SystemC. Ces mots clés encapsulent les appels de services du RTOS. L'API renferme également une librairie constituant les méthodes de communication permettant le transfert de messages.

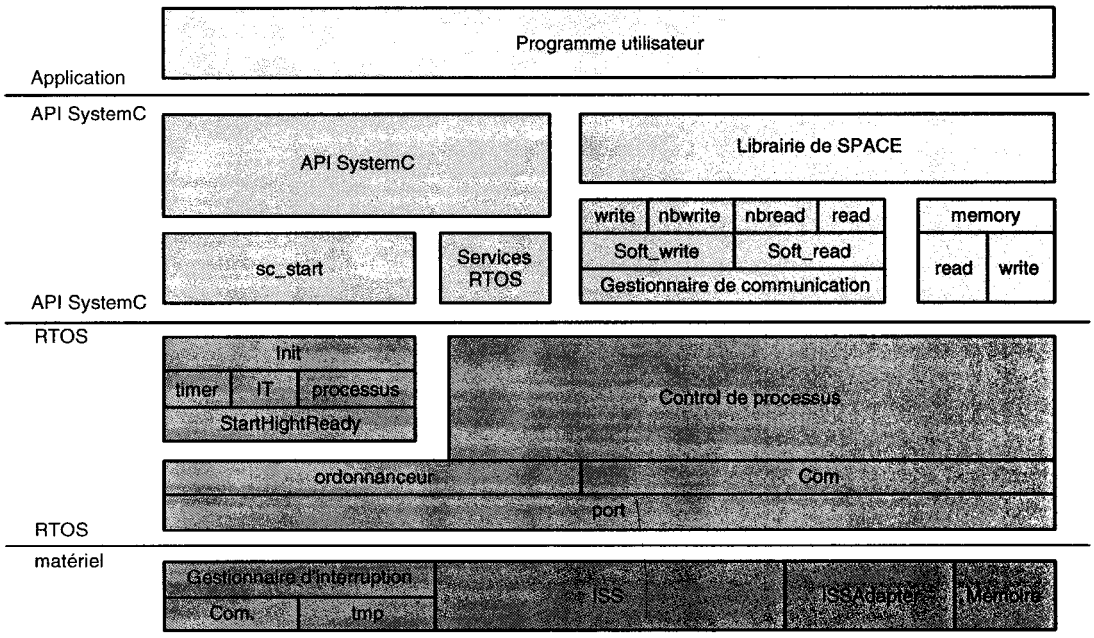

FIGURE 5.1 Structure de la partie logicielle

TABLEAU 5.2 Les fonctionnalités de chacun des niveaux

Niveaux	Partitionnement	RTOS	Communication
L1	non	non	canal de communication haut niveau
L2	oui	RTOS encapsulé dans un processus	Communication IPC sur machine de développement
L3	oui	RTOS exécuté par l'ISS	Modélisation d'un bus en SystemC sur architecture SPACE

5.2 Les parties identiques entre les niveaux

Si les niveaux sont différents, ils possèdent certaines caractéristiques communes. Le tableau suivant présente les différences de chacun des niveaux du point de vue fonctionnel. Ensuite, la partie suivante sera consacrée à la description des fonctions identiques, puis nous verrons ensuite celles qui sont propres à chacun des niveaux L2 et L3. Le niveau L1, moins complexe, sera expliqué dans la section suivante.

5.2.1 L'API SystemC

Portion intégrée d'un code binaire, l'API SystemC sert de médiateur entre deux programmes étrangers. L'utilisation d'une API facilite ici l'interaction entre le système d'exploitation et l'application en SystemC. Elle constitue une couche logicielle indispensable jouant trois grands rôles :

- La fonctionnalité : sa fonction est d'interpréter le code SystemC d'une application afin de convertir les appels de fonctions SystemC en appels de fonctions compréhensibles par le système d'exploitation.
- L'initialisation : elle joue un rôle important durant cette phase d'initialisation

durant laquelle les processus SystemC deviennent des tâches pour le système d'exploitation. Dépendamment du niveau d'abstraction, c'est tout l'environnement du RTOS qui est initialisé permettant à celui-ci de démarrer son ordonnancement.

- La communication : constituée d'un gestionnaire de communication, elle est responsable de la transaction de message d'un module logiciel à un autre, mais également d'un module logiciel vers un module matériel ou inversement.

Nous allons aborder ses différents rôles dans les paragraphes suivants.

5.2.1.1 Les fonctions de la librairie SystemC

L'API fournit les mêmes services que ceux de SystemC : de la construction des modules, des processus, jusqu'aux canaux et ports de communication. Si les appels de fonctions sont identiques, le code interne est différent de la bibliothèque SystemC. En effet, c'est le système d'exploitation qui va prendre la place du simulateur de SystemC, fournissant ainsi aux modules logiciels un ordonnancement logiciel fidèle à la réalité. L'API SystemC fait donc le lien entre les appels de fonctions SystemC et le RTOS. Tous les mots clés de la bibliothèque SystemC ne sont pas encore implémentés dans l'API. Nous allons parcourir la liste des différentes fonctionnalités de la bibliothèque SystemC de manière à connaître celles actuellement supportées par l'API SystemC. Les fonctionnalités non supportées pourront faire l'objet d'une évolution future. En effet, l'API SystemC doit être compatible avec la bibliothèque SystemC.

Les types de données :

SystemC fournit plusieurs types de données, en plus de ceux du C++. Ces nouveaux types de données sont mieux adaptés pour la modélisation matérielle. En effet, il

TABLEAU 5.3 Les types spécifiques fournis par SystemC

Types	Classes	Fonction
Scalaire	sc_bit, sc_logic	Manipulation de bits.
Flottant	sc_fix, sc_ufix	Manipulations de de nombres flottants.
Entier	sc_int, sc_uint, sc_bigint, sc_biguint	Manipulation d'entier jusqu'a 64 bits.
Vecteurs	sc_bv, sc_logic	Manipulation d'ensemble de bits.

devient possible de représenter une donnée avec des types de toutes les tailles et de manipuler plus facilement un bit particulier. La manipulation est rendue très aisée grâce à des méthodes spécifiques et la surcharge des opérateurs. Le tableau 5.3 suivant présente les différents types d'opérateurs fournis par SystemC.

Ces différents types ne sont actuellement pas tous supportés par l'API. Notez que la présence dans l'API de ces types de données, plus adaptés à la modélisation matérielle, est discutable. Néanmoins, afin de garantir un partitionnement de l'application, les modules SystemC devront être fonctionnels aussi bien dans la partie matérielle que dans la partie logicielle de l'application. C'est pourquoi les types de données plus spécifiques à la modélisation matérielle doivent être interprétés afin qu'un comportement cohérent puisse être simulé au niveau logiciel. Tous les types de processus ne sont pas supportés, c'est ce que nous allons voir dans le paragraphe suivant.

Les processus :

SystemC propose trois types de processus (ou unités d'exécution) : les SC_METHOD, SC_THREAD et SC_CTHREAD. Selon la spécification de SystemC, les processus SC_CTHREAD et SC_THREAD correspondent de près au comportement d'un processus (thread) logiciel. L'API associe donc les processus de type SC_THREAD et

TABLEAU 5.4 Les paramètres de la méthode WAIT

Paramètre	Fonction	exemple
cycle	mise en attente d'un nombre de cycle	wait(500000) // mise en attente de 500000 cycles ou NS
temps	mise en attente d'une période de temps	wait(5, MS) // mise en attente de 5 millisecondes

SC_CTHREAD à des tâches du RTOS. C'est-à-dire qu'ils auront le même comportement que celui d'une tâche MicroC/OS-II. Pour le moment, les processus SC_METHOD ne sont pas implémentés. Ces derniers ne peuvent être interrompus durant leur exécution, c'est à dire qu'ils ne peuvent pas rendre la main au simulateur en cours d'exécution. En effet, les SC_METHOD ne peuvent pas rendre la main au simulateur tant que leur exécution n'est pas terminée. Ils ne peuvent donc pas jouer le rôle de tâches pour le RTOS puisqu'une tâche, par définition, ne se termine pas. C'est pourquoi, il serait possible d'envisager une SC_METHOD comme une tâche créée dynamiquement, avec une priorité supérieure à toutes les autres, qui appellerait son " destructeur " une fois son exécution terminée. Ceci correspondrait à son équivalent SystemC.

La fonction `wait()` permet à un processus de se mettre en attente d'une période de temps ou d'un évènement, laissant ainsi place à d'autre processus de s'exécuter. Le tableau 5.4 montre les différents paramètre de la méthode wait :

Lors du passage d'un module matériel en logiciel, il existe des problèmes de perte de précision en terme de temps d'attente d'une tâche. En effet, l'ordonnancement logiciel est différent de l'ordonnancement matériel et peut entraîner une divergence de comportement. Au niveau matériel, tous les processus sont exécutés les uns à la suite des autres à chaque cycle d'horloge. Un processus en attente de X cycles va se

mettre en attente, et ne sera exécuté qu'au Xième cycle suivant. Au niveau logiciel, le temps durant lequel la tâche reste en attente n'est pas déterminé seulement par la tâche elle-même mais également par le temps d'exécution et les caractéristiques des autres tâches. Reprenons le cas de notre processus en attente de X cycles. A la fin du Xième cycle, le processus est prêt à être exécuté, mais cela ne garantit pas qu'il sera exécuté. En effet, si une tâche de priorité supérieure est en train de s'exécuter, notre tâche restera en attente d'exécution, aussi longtemps que l'exécution de cette tâche sera terminée. De plus, si d'autres tâches de priorité supérieure sont en attente d'exécution, notre tâche restera en attente.

Dans le cas où le temps d'attente placé en paramètre de la méthode wait() est inférieur à la période d'interruption de l'horloge temps réel (interruption générée périodiquement permettant un changement de contexte), le réveil du processus n'est pas non plus simplement dépendant du processus lui-même. La tâche sera exécutée si la tâche en cours d'exécution redonne le contrôle à l'ordonnanceur et si aucune tâche en attente n'a une priorité supérieure à notre tâche en attente.

Ceci peut donc changer le comportement de l'application puisque certain processus ne s'exécuteront pas à la période désirée, ne satisfaisant pas ainsi les contraintes du système. Les résultats de simulation seront d'une aide précieuse permettant de distinguer les modules pouvant être logiciels de ceux, plus critiques qui devront rester matériels.

Convention spécifique pour le partitionnement :

Tous les modules devront donc être conformes à un " standard " afin de permettre le partitionnement. Un module de l'application, qu'il soit matériel ou logiciel sera constitué d'un seul et même port permettant la communication avec l'extérieur. Celui-ci permettra de " connecter " le module à la partie logicielle ou à la partie

matérielle. Toutes les communications en provenance ou à destination de ce module, devront utiliser le canal spécifique utilisant les méthodes de lecture ou d'écriture qui lui sont propres. Ce port de communication est spécifique à l'architecture SPACE. Les sections suivantes présentent les différentes méthodes permettant la communication du module avec le reste du système par l'intermédiaire de ce canal. Une autre spécification réside dans le fait qu'un module doit posséder une seule étiquette unique, ceci permet de conserver la communication entre les modules, qu'ils soient logiciels ou matériels. De plus, du point de vue logiciel, cette étiquette unique représente également la priorité du module, et donc, du processus puisque, actuellement, un module encapsule un seul et même processus. Cette étiquette pourra à l'avenir être paramétrée dynamiquement afin de pouvoir changer facilement la priorité d'un module entre deux évaluations de partitionnement.

5.2.1.2 L'initialisation pour l'ordonnancement

L'initialisation est une phase très importante puisqu'elle permet de préparer tout l'environnement du système d'exploitation sur lequel l'application cliente va évoluer. Dans SystemC, l'initialisation permet de construire la liste des processus faisant partie de toute la hiérarchie de modules constituant l'application. La pile de processus ainsi créée va être utilisée par le simulateur afin d'exécuter chacun d'entre eux, les uns à la suite des autres suivant leur état (prêt à être exécutée ou en attente). Dans MicroC/OS-II, l'initialisation est regroupée dans une fonction appelée avant toute création de tâche et bien sûr avant le démarrage de l'ordonnanceur (initialisation du contexte de la tâche, de la table des priorités, du mécanisme de gestion des tâches en attentes et prêtes à rouler, des listes d'événements, des queues de messages).

L'API s'inspire donc de ses deux phases d'initialisation. Tout d'abord, c'est la partie

initialisation de SystemC qui est appelée, jusqu'à l'obtention de la pile d'initialisation des processus. Ensuite, c'est l'initialisation de MicroC/OS-II qui est appelée. Chaque processus va engendrer la création d'une tâche MicroC/OS-II. Puis, enfin, la première tâche va pouvoir s'exécuter et l'ordonnanceur du système d'exploitation va être appelé. Celui-ci va identifier la première tâche à exécuter puis, de ce fait, mettre fin au processus d'initialisation.

Le schéma 5.2 illustre les différentes fonctions jouant un rôle dans l'initialisation du système.

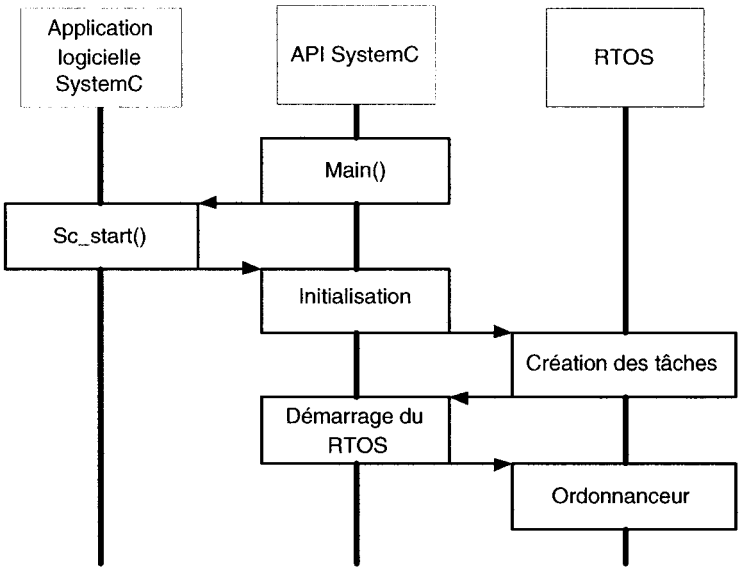

FIGURE 5.2 Initialisation de la partie logicielle

L'ordonnanceur de SystemC est capable de reproduire le comportement d'un système matériel où les processus semblent s'exécuter à l'intérieur d'une seule et même période de temps, donnant ainsi l'impression de parallélisme. Le simulateur de SystemC est donc très proche des simulateurs VHDL pour l'ordonnancement des modules matériels. Au niveau logiciel, l'ordonnancement est différent. Un processeur

ne peut pas exécuter tous les processus en même temps. Ceux-ci doivent s'exécuter de façon concurrente. Un seul processus est exécuté à chaque instant t par le processeur. Chacun des processus peut être interrompu pour laisser place à l'exécution d'un autre ou de plusieurs autres processus avant de revenir à l'exécution du processus précédent. Ceci donne l'impression que tous les processus évoluent en même temps.

La plupart des systèmes d'exploitation temps réel possèdent un ordonnancement basé principalement sur la priorité. Bien sûr, d'autres ordonnancements existent, certain RTOS combinent plusieurs mécanisme d'ordonnancement, mais pour ce travail, nous nous sommes limités au RTOS utilisé qui est basé simplement sur les priorités. Avant l'exécution d'une tâche, l'ordonnanceur choisit celle dont la priorité est supérieure aux autres. Périodiquement, le système vérifie si une tâche de priorité supérieure n'est pas prête à être exécutée. Si tel est le cas, la tâche en cours d'exécution est interrompue puis mise en attente, afin de laisser place à la tâche de priorité supérieure. Dans le cas contraire, la tâche en cours continue son exécution. Un tel système est dit préemptif.

L'ordonnancement de SystemC n'est pas basé sur les priorités. Il ne renferme également aucun mécanisme de préemption. En effet, les tâches s'exécutent les unes à la suite des autres de manière séquentielle et ne peuvent pas être interrompues. La tâche décide par l'appel d'une fonction (la fonction `wait()`), de se mettre en attente et ainsi de rendre le contrôle au simulateur de manière à ce qu'il exécute la prochaine tâche prête à être exécutée.

Afin que le système d'exploitation puisse avoir une information sur la priorité de la tâche qu'il va devoir ordonnancer, il est nécessaire d'ajouter cette nouvelle fonctionnalité dans le design des modules logiciels. L'API SystemC devra être en mesure de fournir ainsi toutes les informations nécessaires à la création d'une tâche par

l'OS. Actuellement, la priorité d'un processus est attribuée lors de sa création et n'est donc pas modifiable en cours d'exécution. La priorité est déterminée par une étiquette du module. Puisque chacun des modules a une étiquette unique, il a donc une priorité unique. Ceci est, pour le moment, une restriction puisqu'il n'est donc pas possible d'ordonnancer des modules ayant des priorités identiques. D'ailleurs, cette fonction n'est toutefois pas possible avec le RTOS utilisé (MicroC/OS-II). De plus, selon les spécifications actuelles de SPACE au niveau des communications, la gestion des modules ayant les même étiquettes n'est pas encore implémentée. Ceci pourra faire l'objet d'une évolution future en dissociant la priorité de l'étiquette du module.

5.2.1.3 La communication

Comme nous l'avons vu précédemment, l'API SystemC fournit un port de communication permettant de connecter les modules logiciels avec la partie logicielle de la plateforme de partitionnement, quel que soit le niveau d'abstraction. Le composant chargé de la gestion des communications logicielles est le gestionnaire de messages. Il acheminera le message en provenance d'une tâche logicielle ou matérielle vers un mécanisme de liste gérant des boîtes de réception de la tâche destinataire. Le gestionnaire de messages joue donc le rôle d'un centre de trie de messages de manière transparente pour l'application contenant les modules logiciels. Cette gestion est basée sur des listes STL et pourra donc être optimisée afin d'améliorer les performances et surtout afin d'utiliser un mécanisme plus déterministe. La figure 5.3 présente les différents blocs constituant l'API SystemC.

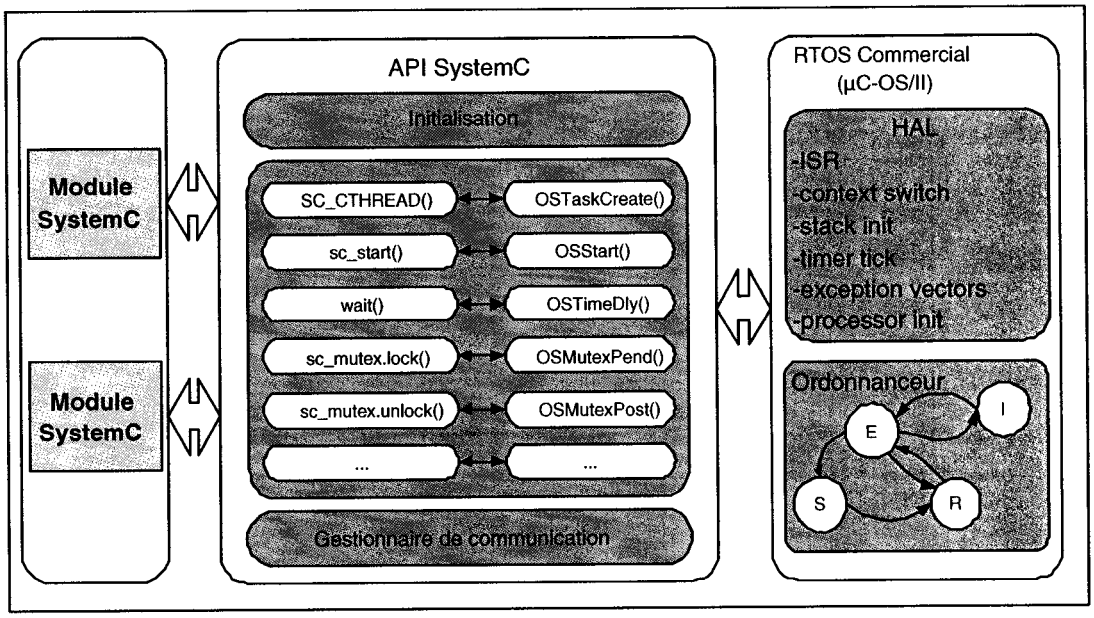

FIGURE 5.3 API SystemC

Le tableau 5.5 présente les différentes fonctions supportées par le gestionnaire de communication (appelé aussi gestionnaire de messages). Le gestionnaire de messages est constitué de plusieurs listes :

- Une liste pour les tâches logicielles[1] contenant les étiquettes de chacune des tâches logicielles ainsi que pour chacune d'entre elles, un pointeur sur une liste de tâches qui ont envoyé un message à cette tâche.
- Une liste des tâches envoyeurs de messages (boîte de réception). Ceci permet de

TABLEAU 5.5 Les méthodes de communication

Méthodes	Fonctions
nb_write	Écriture non bloquante (non synchronisée sur la lecture)
nb_read	Lecture non bloquante (non synchronisée sur l'écriture)
write	Écriture bloquante (en attente de la lecture)
read	lecture bloquante (en attente de l'écriture)
mem_write	Écriture à une adresse en mémoire
mem_read	Lecture à une adresse en mémoire

trier les messages reçus, par envoyeurs. Cette liste contient des identifiants des tâches ainsi qu'un pointeur sur le message à transmettre.

- Une liste contenant les messages rangés par ordre d'arrivée. Le premier message entré dans la liste sera le premier à être lu et donc à être retiré de la liste (FIFO).
- Une liste de transactions permet d'identifier les transactions en cours. Cette liste est complètement indépendante des listes précédentes. Elle est utilisée pour la synchronisation dans les communications bloquantes. Cette liste tient à jour les transactions en cours entre deux tâches lorsqu'une des deux est en attente de l'autre (aussi bien pour une lecture que pour une écriture). La mise en attente des tâches est également gérée par cette liste grâce à des sémaphores fournis par le RTOS. Une fois la transaction terminée, celle-ci est retirée de la liste.

La figure 5.4 montre les différentes listes mise en jeu dans la gestion des communications.

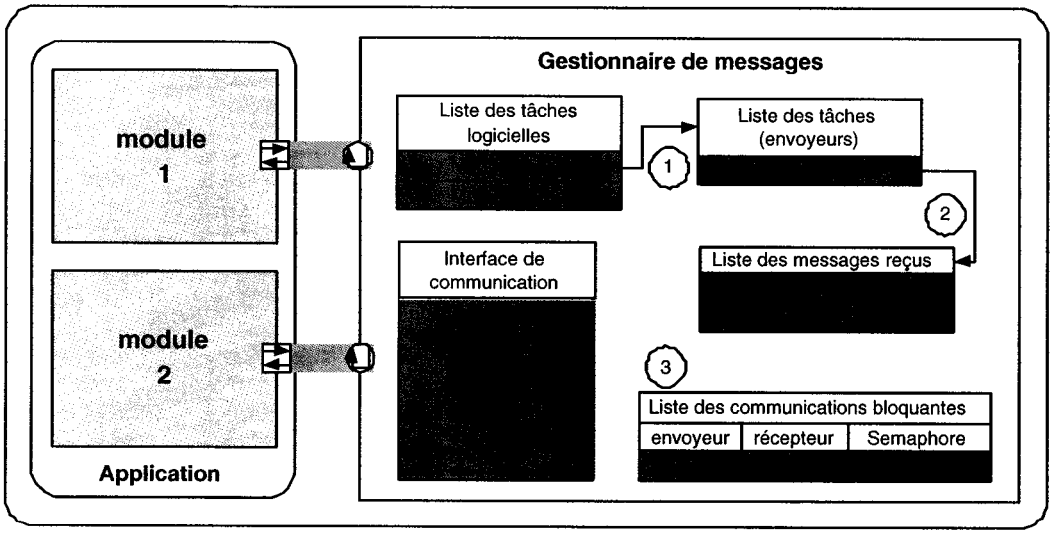

FIGURE 5.4 Organisation des listes pour la communication

Nous allons voir comment un module logiciel peut envoyer et recevoir un message en provenance d'une tâche logicielle. Ces fonctionnalités sont identiques pour les niveaux L2 et L3. Les communications entre les parties logicielles et matérielles seront détaillées en fonction du niveau d'abstraction dans les sections suivantes.

Réception d'un message en provenance d'une tâche logicielle

La fonction (`nb_read()` de l'anglais *non blocking read*) permet à un module logiciel d'interroger sa boîte de réception (de manière non bloquante) pour savoir si elle a reçu un message. Afin d'expliquer les différentes étapes de cette communication, l'organigramme présenté à la figure 5.5 illustre les explications suivantes. De plus, la figure 5.4 nous fournit un exemple dans lequel un module logiciel (module 2) envoie un message à un autre module logiciel (module 1).

- La tâche *module 1* envoie la requête de lecture non bloquante au gestionnaire de communication.
- Le gestionnaire identifie la tâche *module 1* se trouvant dans la liste des tâches logicielles puis il obtient le pointeur sur la liste des tâches qui ont envoyé un message à cette tâche.
- Il parcourt la liste à la recherche de l'étiquette de l'envoyeur *module 2*.

Si l'étiquette est trouvée, le gestionnaire des communications récupère le message de la liste de réception (2) et copie son contenu à l'adresse de destination (pointée par la requête). Il interroge également la liste des transactions (3) afin de débloquer la tâche *module 2* éventuellement en attente par une écriture bloquante (`write()`). Si la recherche n'a pas été satisfaisante (étiquette non trouvée (1)), la tâche *module 1* est avertie qu'aucun message de la part de *module 2* ne lui est adressé.

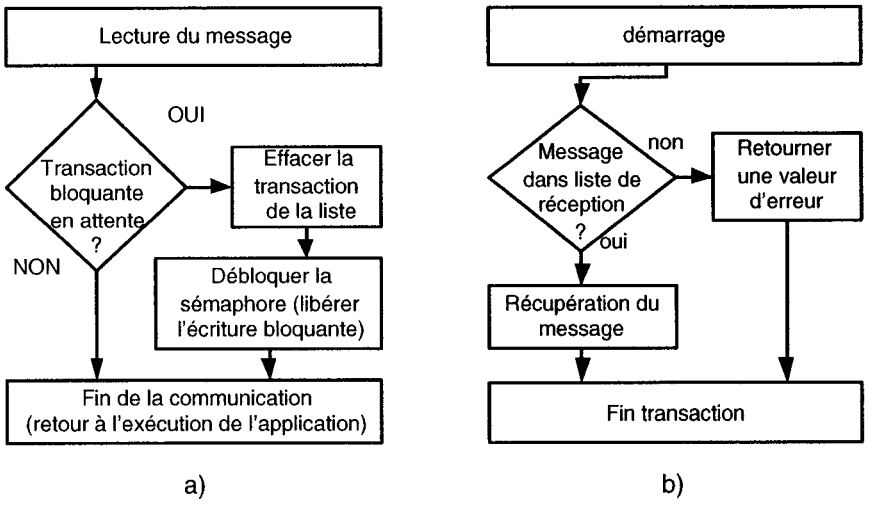

FIGURE 5.5 a)Lecture non bloquante, b)Lecture générique appelée par a).

L'interrogation de la boite de réception peut se faire de manière bloquante par l'intermédiaire de la fonction `read()`. Dans le cas où aucun message en provenance de la tâche *module 2* et à destination de la tâche *module 1* n'est disponible (étiquette non trouvée (1)), celle-ci sera mise en attente, bloquée par un sémaphore (fourni par le RTOS). La tâche *module 1* restera en attente aussi longtemps que le sémaphore ne sera pas débloqué (3) par l'arrivé d'un message envoyé par *module 2.* La figure 5.6 permet d'illustrer ce fonctionnement expliqué précédemment.

Envoie d'un message à destination d'un module logiciel

La fonction (`nb_write()` de l'anglais *non blocking write*) permet à un module logiciel d'envoyer un message dans la boîte de réception d'un autre module (de manière non bloquante). Afin d'expliquer les différentes étapes de cette communication, l'organigramme présenté à la figure 5.7 va nous permettre d'illustrer les explications suivantes. De plus, la figure 5.4 nous fournit un exemple où un module logiciel (module 2) envoie un message à un autre module logiciel (module 1).

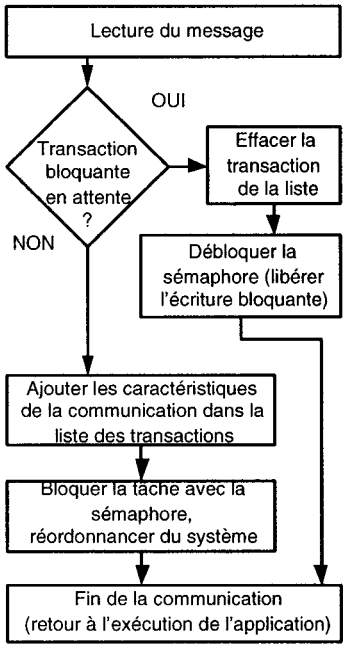

FIGURE 5.6 Lecture bloquante utilisant la lecture générique (b) de la figure 5.5.

- La tâche *module 2* envoie la requête d'écriture non bloquante au gestionnaire de communication.
- Le gestionnaire identifie la tâche destinataire (*module 1*) se trouvant dans la liste des tâches logicielles. Si celle-ci n'est pas contenue dans cette liste, la tâche destinataire se trouve dans la partie matérielle. Le mécanisme de communication sera expliqué dans les sections portant sur la communication au niveau 2 et au niveau 3.
- Le gestionnaire ajoute dans la liste, l'étiquette de la tâche *module 2* (1) puis, le message à transmettre est ajouté dans la boîte de réception (liste de messages reçus (2)).
- Il interroge également la liste des transactions (3) afin de débloquer la tâche *module 1* éventuellement en attente par une lecture bloquante (`read()`).

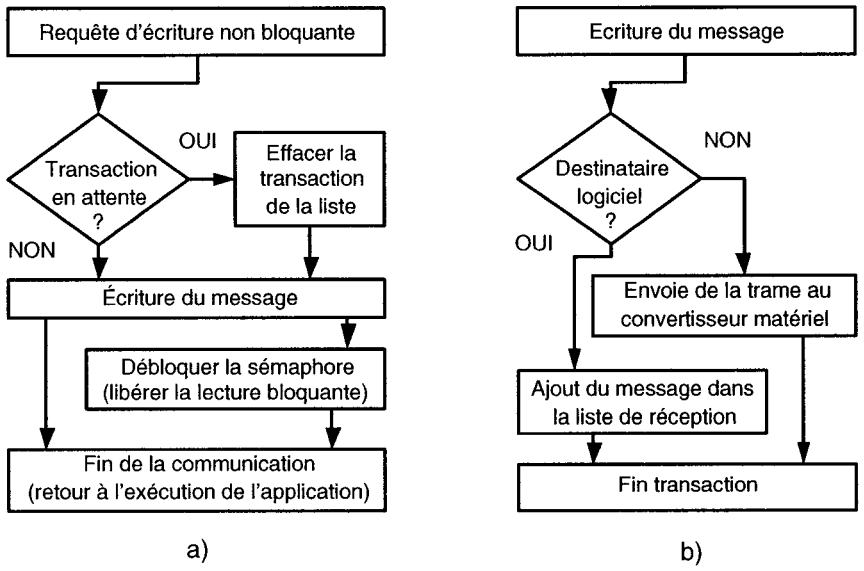

FIGURE 5.7 a)Ecriture non bloquante, b)Ecriture générique appelée par a).

L'envoi d'un message peut se faire de manière bloquante par l'intermédiaire de la fonction `write()`. La figure 5.8 permet d'illustrer le fonctionnement. Si aucune transaction en provenance de la tâche *module 2* et à destination de la tâche *module 1* n'est disponible (liste des transactions (3)) :

- Le gestionnaire de communication ajoute la transaction dans la liste des transactions en cours.
- La tâche *module 2* est mise en attente, bloquée par un sémaphore (fourni par le RTOS).
- Cette dernière restera en attente aussi longtemps que le sémaphore ne sera pas débloqué (3) par l'arrivé d'une requête de lecture de la tâche *module 1*. La figure 5.8 permet d'illustrer ce fonctionnement.

Actuellement, le développement de l'API SystemC est rudimentaire. Celle-ci ne possède pas tous les mots clés de la librairie SystemC. Il n'est donc pas 100% compa-

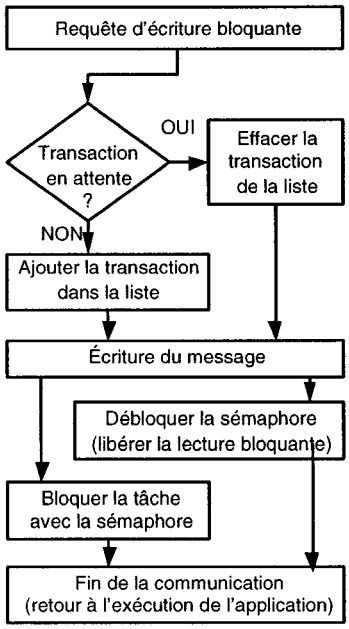

FIGURE 5.8 Ecriture non bloquante

tible. Cependant, celle-ci contient une base permettant la construction d'exemples simples montrant les fonctionnalités de la plate-forme d'aide au partitionnement. Nous allons donc détailler tout d'abord les différentes fonctions supportées par cette API. Puis ensuite, nous parlerons du gestionnaire de communication qui joue un rôle indispensable. Nous allons nous attarder à l'initialisation qui va nous permettre de comprendre comment le système fonctionne.

5.3 Les spécificités du niveau L2

5.3.1 Le RTOS

MicroC/OS-II a été porté pour Linux. Le support (Hardware Abstraction Layer) permettant au RTOS de fonctionner utilise les fonctions de sauvegarde et de restauration de pile propres à Linux pour le changement de contexte. De plus, il utilise une horloge de Linux qui permet de générer une interruption périodiquement de

manière à émuler une horloge matérielle. Ce support logiciel est très réduit, mais fournit le minimum nécessaire à l'exécution du RTOS.

5.3.2 La communication logicielle/matérielle

Nous allons expliquer comment fonctionne la communication entre un module logiciel et un module matériel lors d'une écriture et d'une lecture au niveau L2. Nous avons vu dans le chapitre de la méthodologie (section 4.4) que cette communication est basée sur la synchronisation des messages et utilise un mécanisme de communication entre processus fourni par le système d'exploitation. Cette solution a permis d'évaluer très rapidement la fonctionnalité du système mais ne s'est pas révélée très performante au niveau des résultats de simulation. Elle pourra être optimisée dans une version future.

Envoie d'un message à destination d'un module matériel

Lorsqu'un module veut envoyer un message à un autre module. Le gestionnaire vérifie l'étiquette du module destinataire. Si celui-ci ne figure pas parmi la liste des modules logiciels, le destinataire se trouve donc dans la partie matérielle. La procédure pour acheminer ce message sera différente du cas où le destinataire est logiciel (présenté à la section 5.2.1.3).

Rappelons que ce niveau d'abstraction est composé de deux processus. Un processus (que nous appellerons processus matériel) exécute le simulateur de SystemC ordonnançant la partie matérielle de l'application partitionnée. Le deuxième processus est divisé en deux tâches (thread). L'une exécute la partie logicielle, elle est constituée du RTOS (MicroC/OS-II), de l'API SystemC et de l'application écrite en SystemC. L'autre tâche est chargée de la communication, elle répond aux requêtes du processus de la partie matérielle.

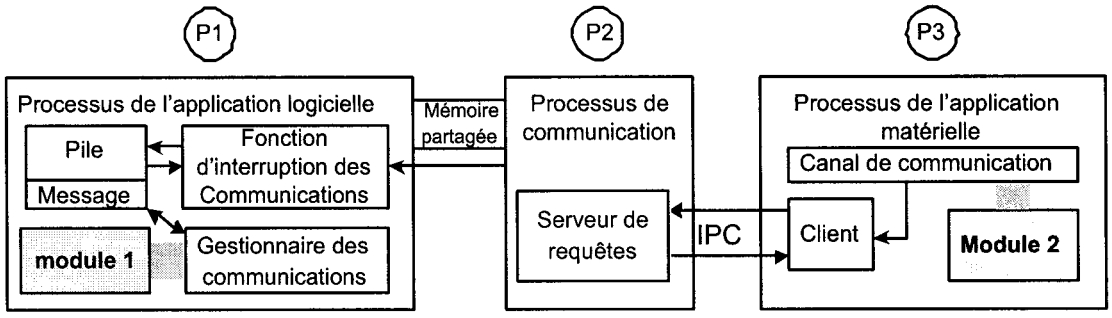

FIGURE 5.9 Communication logicielle/matérielle au niveau L2

L'envoi d'un message à partir d'un module logiciel se fait de manière non bloquante. C'est à dire que le module logiciel désireux d'envoyer un message à une tâche matérielle ne sera jamais bloqué en attente de la requête de la part du processus matériel. Le message sera conservé de manière temporaire jusqu'à ce qu'un module matériel envoie une requête de lecture au processus de communication logiciel. Celui-ci identifie la requête, puis enverra la requête dans un pipeline chargé de la communication de la tâche de communication vers la tâche exécutant la partie logicielle de l'application. Ensuite, un signal sera déclenché afin d'interrompre la tâche logicielle afin d'exécuter la fonction permettant de récupérer la requête présente dans le pipeline. Après interrogation du gestionnaire de messages de l'API SystemC, le message à destination de la partie matérielle (toujours en attente) est encapsulé dans la requête de réponse, puis renvoyé dans un pipeline chargé des communications de la tâche logicielle vers la tâche des communications. Le RTOS de la tâche logicielle reprends alors son ordonnancement. La tâche des communications lit ensuite la réponse de la requête présente dans le pipeline correspondant, puis construit une requête de réponse et l'envoi au processus matériel toujours en attente. La figure 5.9 permet d'illustrer ce qui a été expliqué précédemment.

Réception d'un message en provenance d'un module matériel

Du point de vue d'un module logiciel, le mécanisme de réception d'un message en provenance d'un module matériel ou logiciel est identique. C'est-à-dire qu'il interroge sa boîte de réception présente dans le gestionnaire des communications situé dans l'API SystemC (voir section 5.2.1.3 : Réception d'un message en provenance d'une tâche logicielle). Concernant le mécanisme interne au gestionnaire des communications est quant à lui un peu différent. En effet, lorsque le processus matériel (P3) envoie une requête d'écriture au processus de communication (P2) en utilisant un protocole basé sur les *sockets*, ce dernier identifie la requête et interrompt le processus de l'application logicielle (P1) en lui envoyant un signal POSIX. La routine d'interruption achemine le message au gestionnaire des communications de l'API qui copiera ensuite le message dans la boite de réception de la tâche destinataire. Le gestionnaire de l'API débloque la tâche de lecture si celle-ci était en attente, puis, retourne une réponse au processus de l'application matérielle (P3) afin de le débloquer (figure 5.9).

Nous venons de voir deux types de requêtes qui transitent du processus matériel vers le processus logiciel. Il existe également deux autres requêtes. Une requête permet d'obtenir la liste des identifiants des modules logiciels. En effet, lors de l'initialisation du coté logiciel celui-ci établit, comme nous l'avons vu précédemment, la liste des tâches logicielles. Cette liste sera envoyée en réponse à la requête matérielle. La dernière requête permet de terminer la simulation par l'envoi d'un message de fin au processus logiciel.

Pour le moment, il n'est pas possible pour un module logiciel de communiquer avec la mémoire qui se trouve dans le processus matériel. Les fonctions qui permettent d'écrire et de lire des données à une adresse en mémoire ne sont donc pas utilisables.

5.4 Les spécificités du niveau L3

5.4.1 Le RTOS

Rappelons que l'architecture du niveau L3 est équipée d'un émulateur de processeur. Celui-ci est basé sur un ISS (Instruction Set Simulator) reproduisant le comportement d'un ARM7. Cette gamme de processeurs ARM ne comporte pas d'unité de gestion de la mémoire (*Memory Management Unit*). L'espace mémoire est entièrement accessible par les tâches de l'application. Il n'existe aucune protection mémoire aussi bien en lecture qu'en écriture. Notons que la plate-forme ne peut donc pas supporter les systèmes d'exploitation requérant ce type de gestion mémoire (comme Linux par exemple).

Ici, MicroC/OS-II a été porté afin de pouvoir être exécuté sur le processeur de type ARM. En plus des modifications propres à l'initialisation des piles des tâches et des fonctions de changement de contexte, plusieurs fonctions ont été ajoutées de manière à initialiser certains périphériques propres à la plate-forme, comme l'horloge temps réel et le gestionnaire d'interruptions. Ces fonctions se trouvent dans la couche logicielle appelée HAL [34] (Hardware Abstraction Layer). Nous n'allons pas détailler toutes les caractéristiques de cette couche logicielle, mais simplement parcourir les différentes fonctionnalités les plus importantes.

L'initialisation des piles est très importante lors du démarrage du système puisque ses adresses d'initialisation déterminent où doit se rendre le processeur lors d'un changement de mode. Par exemple, quand une interruption intervient, le processeur change de mode pour passer du mode superviseur (SVC) au mode exception (IRQ). Il permet également de déterminer le mode d'exécution. Rappelons simplement que le processeur de type ARM7 comprend sept modes d'exécution. Celui utilisé ici est le mode superviseur (SVC). Pour plus d'informations sur le processeur ARM, voir

la référence [1].

Le contexte d'une tâche regroupe toutes les informations nécessaires sur l'état des registres du processeur, juste avant un changement de contexte (la mise en attente d'une tâche en cours d'exécution pour activer une tâche prête à être exécutée). Ces fonctions de sauvegarde et de restauration des registres sont dépendantes du processeur, elles sont spécifiques au type de processeur utilisé.

Toute architecture à base de processeur comporte une horloge. L'horloge utilisée pour l'architecture SPACE est un module SystemC générant de manière périodique des interruptions aux processeurs. Ce module est paramétrable et permet de configurer, avant le démarrage, la période des interruptions.

Le gestionnaire d'interruptions est également un module SystemC qui permet de générer une interruption à l'ISS lorsqu'une de ses entrées est activée (figure 3.3). Il joue le rôle de multiplexeur puisque le processeur ne comprend qu'une seule entrée pour les interruptions. Ce périphérique ne comprend aucune gestion de priorité, cette fonction est laissée au logiciel. Lorsqu'une interruption intervient, l'ISR (la fonction de traitement des interruptions) vient lire un registre du gestionnaire d'interruptions pour connaître l'identité de l'interruption déclenchée et effectuer le traitement qui s'impose. Puis l'interruption est acquittée afin que le gestionnaire puisse se mettre en attente d'une prochaine interruption.

5.4.2 La communication logicielle/matérielle

Nous allons expliquer comment fonctionne la communication entre un module logiciel et un module matériel lors d'une écriture puis d'une lecture au niveau L3, avec l'architecture SPACE.

Envoi d'un message à destination d'un module matériel

Nous avons vu que, lors d'une écriture de la part d'un module logiciel, le gestionnaire de messages interroge la liste des étiquettes logicielles afin de savoir si le module destinataire s'y trouve. Dans le cas contraire, où la tâche destinataire ne se trouve pas parmi les tâches logicielles, le message est alors destiné à un module matériel. Le gestionnaire de message s'adresse à un module matériel spécifique afin d'envoyer le message à la partie matérielle qui sera chargée d'acheminer celui-ci au module destinataire. Ceci s'effectue en envoyant un paquet prédéfini comportant l'identifiant du destinataire, celui de l'envoyeur ainsi que la taille du message puis enfin, le message lui-même. Le module matériel spécifique à SPACE est appelé Adaptateur ISS (section 3.2.5). Il est chargé de la recomposition du message pour l'envoyer par le bus matériel vers le module destinataire. La figure 5.10 montre le lien entre les différents modules. Elle montre que toute la partie logicielle est exécutée par le ARM.

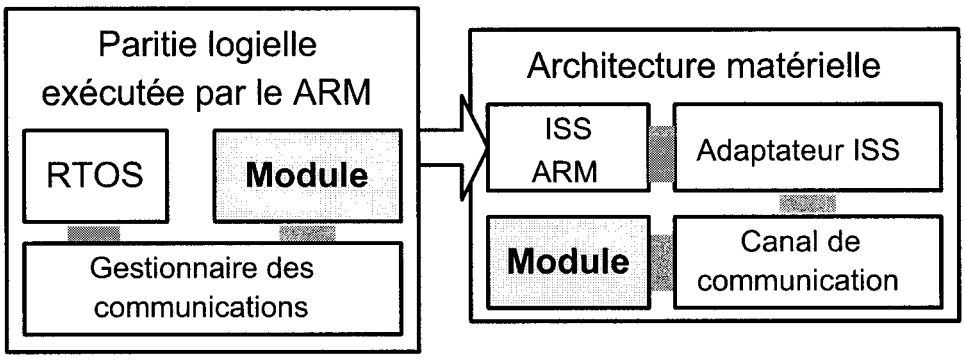

FIGURE 5.10 Communication logicielle/matérielle au niveau L3

Réception d'un message en provenance d'un module matériel

Lorsqu'un module matériel envoie un message à un module logiciel, le décodeur (adaptateur ISS) génère une interruption. L'ISS interrompt l'exécution et active une fonction chargée de récupérer le message en provenance du décodeur. Le gestionnaire des communications de l'API SystemC acheminera le message jusqu'à la boîte de réception du destinataire. La figure 5.11 illustre le fonctionnement de ce mécanisme :

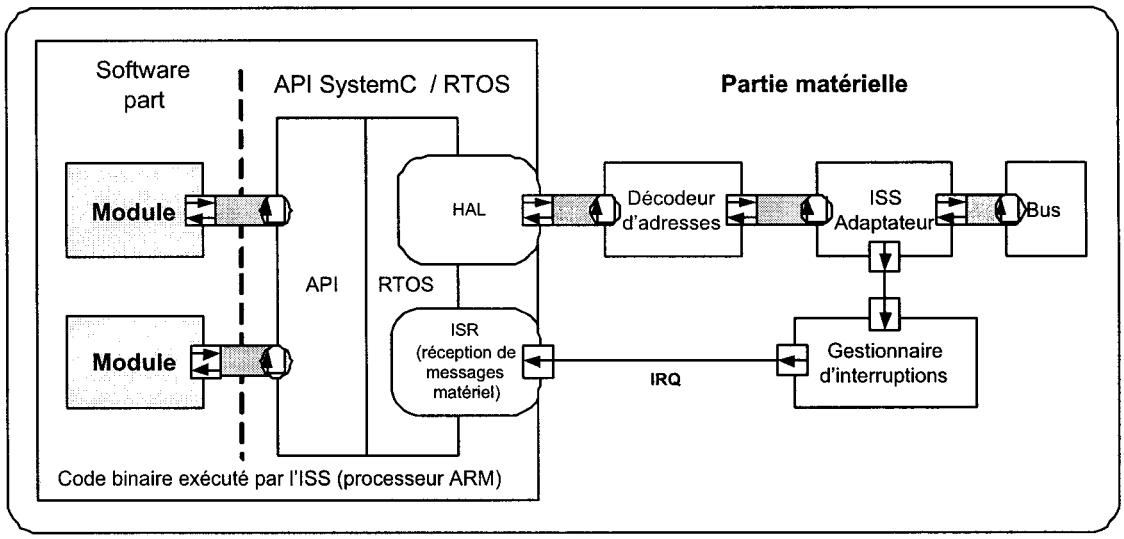

FIGURE 5.11 Fonctionnement de la méthode de réception matérielle

L'écriture et la lecture en mémoire

Des fonctions permettent également d'écrire et de lire des données à une adresse en mémoire. Un module peut donc utiliser une communication par adresses, peu importe sa nature, qu'il soit matériel ou logiciel. La mémoire est partagée entre tous les modules de l'application. Comme nous l'avons vu précédemment, il n'existe

aucune protection mémoire (pas de MMU).

La méthode de lecture concerne la lecture à une adresse en mémoire (mem_read) (tableau 5.5). Cette méthode permet de lire un message d'une longueur donnée en paramètre à une adresse. Ceci est réalisé en copiant le message à partir de l'adresse passée en paramètre vers un pointeur de destination (lui aussi, passé en paramètre).

La méthode d'écriture concerne l'écriture à une adresse en mémoire (mem_write). Cette méthode permet d'écrire à une adresse en mémoire (passée en paramètre) un message dont le pointeur ainsi que la taille sont passés en paramètre.

5.5 Récapitulatif

Nous venons de parcourir le fonctionnement de la partie logicielle du système, de l'initialisation en passant par l'interprétation des mots clés SystemC jusqu'à la communication. Tout ceci est renfermé dans une API constituée de librairies qui, accompagnées du RTOS et de l'application, forment un seul et même fichier binaire. Ce fichier binaire sera exécuté en tant que processus par exemple avec le système d'exploitation Linux pour le niveau L2 et par le processeur de la plate-forme SPACE pour le niveau L3.
Dans le prochain chapitre, nous allons présenter les tests ainsi que les résultats obtenus.

CHAPITRE 6

RÉSULTATS ET DISCUSSION

Le chapitre suivant présente deux exemples mettant en application la méthodologie présentée précédemment. Ces tests ont également pour objectif la validation des fonctionnalités de l'outil de partitionnement. Un premier exemple permettra d'utiliser les trois niveaux de raffinement logiciel. Cet exemple, constitué de quatre modules communicants entre eux, a été développé pour valider le niveau L2 de la méthodologie. Ensuite, nous ferons varier le partitionnement d'une application à travers un second exemple plus complexe que nous simulerons au troisième niveau (L3). Cet exemple plus réaliste décrit une application de traitement de données audio ou vidéo. Nous détaillerons trois versions d'un partitionnement utilisant le niveau L3 : une version entièrement logicielle, une autre entièrement matérielle, puis enfin une version partitionnée. Cet exemple a fait l'objet d'un article de conférence [6].

Actuellement, la procédure de passage d'un niveau à l'autre n'est pas automatisée et ne renferme aucun algorithme intelligent capable de déterminer le partitionnement idéal. Pour chaque configuration, l'utilisateur démarre la simulation. Le nombre de cycles d'horloge total parcouru ainsi que la durée de la simulation sont enregistrés. Ainsi, le partitionnement optimal estimé est celui où la configuration a donné des temps de simulation satisfaisant les contraintes de temps tout en minimisant le nombre de modules matériels (afin de réduire les coûts).

6.1 Exemple producteur/consommateur

La figure 6.1 décrit l'exemple d'une application producteur/consommateur qui a été utilisé pour l'évaluation du fonctionnement du niveau L2 de la méthodologie. Cette application est constituée de quatre modules qui s'envoient une information les uns à la suite des autres.

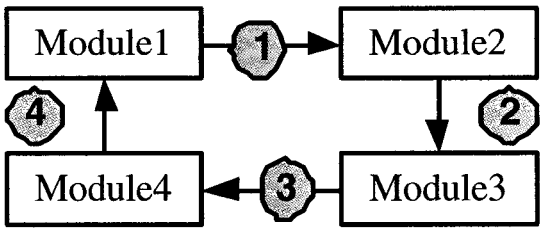

FIGURE 6.1 Exemple : producteur/consommateur

Le module M1 commence par envoyer une donnée au module M2. Cette donnée est alors reçue par le module M3 qui la modifie et l'envoi au module M4. Le cycle se termine par l'envoie d'une donnée du module M4 vers le module M1, laissant place au suivant. Le tableau 6.1 nous présente les résultats de simulation obtenus en fonction des différents niveaux de la méthodologie (L2 et L3) avec pour chacun d'eux, trois versions partitionnées :

- Tous les modules de l'application sont logiciels.
- Tous les modules de l'application sont matériels.
- L'application est divisée en deux : Les modules M1 et M2 sont logiciels tandis que les modules M3 et M4 sont matériels.

Rappelons que le niveau L1 ne propose aucun partitionnement. Cela explique pourquoi une seule version est présente.

TABLEAU 6.1 Résultats : producteur/consommateur

Version	Niveau	Modules logiciels	Modules matériels	Nombre de cycles exécutés	Temps de simulation (sec.)
1	L1	-		N/A	1
2-1	L2	M1, M2, M3, M4	-	N/A	2
2-2	L2	-	M1, M2, M3, M4	N/A	4
2-3	L2	M1, M2	M3, M4	N/A	24
3-1	L3	M1, M2, M3, M4	-	6266236	46
3-2	L3	-	M1, M2, M3, M4	3182	1
3-3	L3	M1, M2	M3, M4	4561980	34

6.1.1 Le niveau L1

Rappelons qu'au premier niveau de raffinement (niveau L1), il n'y a pas d'architecture, tous les modules sont connectés entre eux par un canal d'interconnections et simulés au niveau UTF (*untimed functional*), sans notion de temps, par le simulateur de SystemC. Aucun partitionnement n'est possible à ce niveau puisque l'objectif est la simulation fonctionnelle à haut niveau (non dépendant de l'architecture ni du RTOS). Les résultats de ce niveau L1 correspondent à la version 1. La simulation est très rapide, l'accent est mis sur la fonctionnalité de l'application.

6.1.2 Le niveau L2

Comme nous l'avons vu dans la méthodologie, ce niveau d'abstraction permet d'effectuer un partitionnement. Il existe trois niveaux de partitionnement au niveau L2 présentés dans le tableau des résultats : une version entièrement logicielle (2-1),

une version entièrement matérielle (2-2) et une version partitionnée où les modules M1 et M2 sont logiciels tandis que les modules M3, M4 sont matériels.

A ce niveau, comme au précédent, la synchronisation est également basée sur la communication entre les modules. C'est pourquoi, il n'y a pas d'information concernant le nombre de cycle.

Les modules matériels sont connectés à un canal de communication identique au niveau L1. La seule différence réside dans l'ajout d'un mécanisme de communication (de type IPC) permettant de communiquer avec le processus de communication logiciel. Dans le cas de la version où les modules sont tous matériels (version 2-2 d'après le tableau), les résultats de simulation sont identiques au niveau L1 puisque le mécanisme de communication est le même.

Au niveau L2, les modules logiciels sont connectés à l'API SystemC. C'est à dire qu'ils sont inscrits dans le fichier permettant de recenser les modules de l'application qui sont logiciels. Ces modules, ainsi que le RTOS et l'API SystemC forment ainsi un processus qui sera contrôlé par le processus chargé des communications entre les parties logicielles et matérielles.

La version 2-3 est la version la plus lente en terme de durée de simulation à cause du mécanisme de synchronisation de la communication entre le processus logiciel et le processus matériel. Les deux autres versions ont des temps d'exécution presque similaires (versions 2-1 et 2-2). La différence réside dans l'implémentation du code de SystemC et celle du code de la partie logicielle constituée du RTOS, de l'API SystemC et de l'application. Les résultats du niveau L2 ne sont pas très satisfaisants. En effet, les temps de simulation devraient être beaucoup plus rapides qu'au niveau L3 en raison du niveau d'abstraction éliminant les précisions architecturales au niveau cycle. La synchronisation entre les différents processus semble être le goûlot

d'étranglement. Le mécanisme de communication doit être optimisé en éliminant le processus de communication actuellement chargé du transfert de requêtes entre le processus de la simulation logicielle et celui de la simulation matérielle.

6.1.3 Le niveau L3

Le dernier niveau L3, est également désigné comme le niveau TF (Time Functional). Les modules matériels sont connectés à un canal de communication comportant des spécifications de temps afin de modéliser à haut niveau les temps de communication. Il serait possible de modéliser un véritable système d'interconnexion (bus), mais ceci n'est pas le sujet de ce mémoire. Nous ne rentrerons pas dans les détails concernant la modélisation du canal de communication, mais ceux-ci sont disponibles dans [3]. Retenons simplement que les temps de communication entre les modules matériels ne sont plus négligeables, ils ont donc une incidence sur les résultats au niveau du nombre de cycles.

Concernant les versions 3-1 et 3-3, les modules logiciels sont exécutés par l'ISS, de la même manière qu'ils le seraient sur un véritable processeur. La dernière version (3-3) montre que les modules logiciels et matériels peuvent s'exécuter en parallèle fournissant donc des résultats de simulation intéressants par rapport à la version 3-2. Cette dernière montre des résultats plus performants sans module logiciel. Elle serait, théoriquement, la plus gourmande en terme de coût de production.

6.2 Exemple d'une application de traitement de données audio ou vidéo

Nous allons maintenant décrire un exemple un peu plus complexe que le précédent. Cet exemple nous a permis de valider un certain nombre de fonctionnalités du

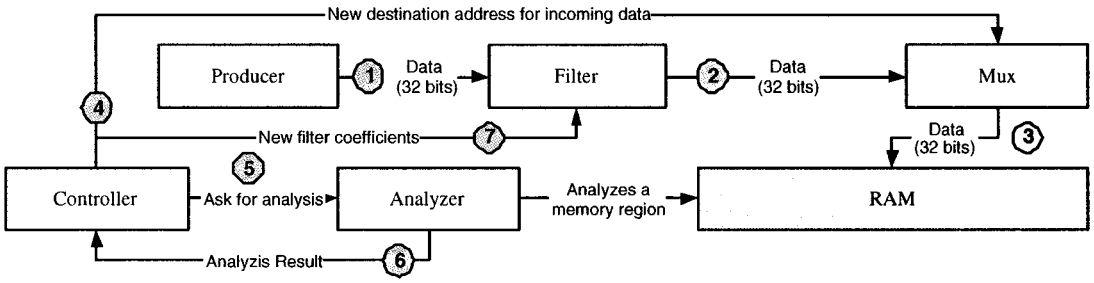

FIGURE 6.2 Exemple de l'article

projet SPACE [6].

Cet exemple, composé de plusieurs modules, représente le traitement d'un flot de données. La figure suivante représente un diagramme fonctionnel montrant les cinq modules constituant notre exemple. Les flèches reliant un module à l'autre représentent les communications.

Chaque module possède une tâche précise, voici les caractéristiques de chacun d'entre eux.

1. Le module "producer" envoie des données sur 32 bits au module "filter"
2. Le module "filter" filtre les données puis les envoient au module "mux".
3. Le module "mux" copie les données à une adresse spécifique en mémoire (RAM).
4. De manière périodique, le module "controller" interrompt le module "mux" afin que celui-ci copie les données dans une autre zone mémoire définie à l'avance.
5. Le module "controller" réveille le module "analyser" pour traiter les donnés qui ont été enregistrées dans la zone mémoire précédemment remplie. Le traitement résulte d'un calcul sur l'ensemble des informations qui servira au module "filtre".
6. Le module "controller" se remet en attente, jusqu'à sa prochaine exécution.

De la même manière que l'exemple précédent, l'application décrite ici a été modélisée en SystemC suivant le standard présenté dans la méthodologie. L'application est ensuite simulée sur l'architecture SPACE (niveau L3). Les différentes versions du partitionnement sont réalisées avec un minimum d'effort puisque, entre deux partitionnements, seules les connections des modules doivent être modifiées. Plus précisément, le fichier chargé du recensement des modules logiciels doit contenir la liste des modules qui seront pris en compte par le gestionnaire des communications logicielles de l'API SystemC. Pour la partie matérielle, le fichier de configuration de l'architecture matérielle contiendra la liste des modules matériels ainsi que les connections au canal de communication. Les parties matérielles et logicielles sont ensuite recompilées et exécutées.

Nous avons effectué trois types de partitionnement :

- La version matérielle où tous les modules sont connectés au canal de communication matériel.
- La version logicielle où tous les modules sont connectés à l'API SystemC du côté logiciel.
- La version partitionnée où les modules "controller" et "analyser" sont connectés à l'API SystemC du côté logiciel. Les autres modules : "producer", "filter" et "mux" sont matériels.

Le tableau suivant présente les résultats de simulation obtenus sur un ordinateur de type Intel Pentium III 600 Mhz avec 128 MB de mémoire). Pour chacune des versions, 1024 entiers sont générés par le module "producer".

Les résultats de la version UTF (1) montrent que la simulation est très rapide au niveau UTF (version 1), puisqu'aucune contrainte de temps n'est spécifiée. Ceci permet la vérification de la fonctionnalité de l'application en faisant abstraction des caractéristiques du mode de communication (temps de transfert d'une donnée d'un module à un autre). Le temps de simulation (nombre de cycle d'horloge)

TABLEAU 6.2 Résultats de simulation

version	niveau	temps de simulation (cycles)	temps de simulation(secondes)
1	L1	2.3245E06	1
2-1	L3 tout matériel TF	4.6912E06	14
2-2	L3 tout logiciel	2019.6E06	6120
2-3	L3 partitionné	5.2897E06	26

n'est cependant pas égal à zéro, puisqu'afin de permettre aux autres modules de s'exécuter, le module "producer" rend la main au simulateur SystemC à chaque fois qu'une donnée est produite. Ceci a pour conséquence de changer de cycle de simulation.

Dans la version TF (2-1), tous les modules de l'application sont dans la partie matérielle. Cela signifie que tous les modules sont connectés à la partie matérielle. Au niveau TF (version 2-1) la communication entre les modules émule un temps de transfert, et donc est coûteuse en terme de temps. Le nombre de cycle est directement affecté par ses temps de communication (se référer à [3] pour plus de détails sur la partie matérielle).

Dans La version logicielle (2-2), tous les modules de l'application sont dans la partie logicielle. Cela signifie que tous les modules sont connectés à la partie logicielle. Cette version 2-1, entièrement logicielle s'exécute sur l'ISS ARM de l'architecture SPACE. Les résultats de simulation montrent que le temps de simulation a augmenté. Conformément à la réalité, l'application logicielle a pris plus de cycles pour exécuter la même fonctionnalité que la version matérielle. En effet, les résultats obtenus ne font que vérifier la rapidité d'exécution des composants matériels par rapport à ceux logiciels. Ils ont néanmoins permis d'établir deux frontières qui déterminent les performances du système. L'application partitionnée ne pourra pas être plus rapide que la version matérielle et moins rapide que la version logicielle.

Si les spécifications ne sont pas comprises à l'intérieur de ses limites, la conception de l'application doit être totalement réétudiée. Dans le cas contraire, l'application peut être optimisée en terme de coût et de performance. Ceci est possible en trouvant une répartition idéale des composants matériels et logiciels de l'application. Rappelons que le matériel est synonyme de performance et le logiciel engendre des coûts moins élevés pour le développement et la maintenance.

Une version partitionnée (2-3) permet de constater une rapidité d'exécution par rapport à la version logicielle précédente. Dans cette version, seuls les modules "controller" et "analyser" sont restés dans la partie logicielle. Les résultats nous montrent tout d'abord que la partie logicielle et la partie matérielle semblent s'exécuter de manière parallèle, comme dans la réalité. Ceci tend donc à réduire le temps de simulation. Nous voyons que, lors du déplacement de deux modules ("controller" et "analyser") dans la partie logicielle et en conservant les autres modules ("Producer", "Filter" et "Mux") dans la partie matérielle, les résultats de simulation sont très positifs. En effet, l'application consomme 87% moins de cycles par rapport à la version logicielle. Avec une répartition de notre application équivalente à 40% logicielle et 60% matérielle, les performances (rapidité d'exécution) sont très proches de la version 100% matérielle. Le partitionnement est donc encourageant. Gardons à l'esprit que l'objectif du partitionnement est de conserver les performances (fixés par les spécifications) tout en réduisant la quantité de composants matériels. Même si l'amélioration des performances était prévisible, cette version partitionnée a permis de quantifier le gain obtenu par la partitionnement d'une application.

6.3 Conclusion

Les exemples présentés ici ne sont pas très complexes. L'objectif principal était de montrer de manière quantitative les avantages du partitionnement au niveau du temps d'exécution d'une application à l'aide de l'outil de partitionnement (SPACE). Nous avons également appliqué la méthodologie proposée dans ce mémoire à travers le premier exemple, en montrant le passage d'un niveau d'abstraction à l'autre. Il est encore très difficile, à ce niveau de développement, de déterminer le partitionnement optimal. Il reste donc de nombreuses fonctions à ajouter et à optimiser, certaines existantes afin d'élargir ses capacités. Nous allons discuter de quelques évolutions et optimisations dans la conclusion de ce mémoire.

CONCLUSION

7.1 Contribution

À travers ce mémoire nous avons montré comment la méthodologie proposée, constituée de trois niveaux d'abstraction pour la partie logicielle, s'intégrait dans un plus vaste projet de conception d'un outil d'aide au partitionnement pour les systèmes sur puce.

Suivant une approche itérative, la méthodologie de raffinement logicielle permet la validation des fonctionnalités d'une application à chacun des niveaux d'abstraction. De la description du système en SystemC, à haut niveau, à l'intégration dans une plate-forme de co-design, la méthodologie de raffinement permet à l'utilisateur de modifier le partitionnement de l'application à tout moment dans le processus de développement. La simulation fournit également des résultats de plus en plus précis d'un niveau à l'autre.

Les résultats de simulation ont montré que le niveau d'abstraction L2 fournissait aux modules logiciels les fonctionnalités d'un véritable RTOS tout en garantissant une rapidité d'exécution. L'intégration d'un véritable RTOS commercial fournit un ordonnancement logiciel idéal très tôt dans le cycle de développement. La simulation du niveau L3 requiert plus de temps de simulation mais fournit néanmoins plus de précision à chaque cycle. En effet, l'utilisation d'un RTOS ainsi que d'un émulateur de processeur (ISS) permettent de reproduire le comportement du système très proche de la réalité, sur une véritable architecture.

Dans la section suivante, nous allons discuter des limites ainsi que des évolutions qui permettraient d'augmenter les fonctionnalités et les performances de l'outil de

partitionnement.

7.2 Limites/optimisation

L'outil de partitionnement n'en est pour le moment qu'à sa première version. Il va donc pouvoir évoluer grâces à ses améliorations. Nous allons parcourir quelques-unes d'entre elles dans les sections suivantes.

7.2.1 La limitation des résultats de simulation

Il n'existe pour le moment aucun outil d'analyse permettant d'obtenir les performances du code de la partie logicielle de l'application. Un tel outil serait très utile également pour l'optimisation du gestionnaire de messages. Actuellement, les seules informations disponibles à la fin de la simulation sont le temps ainsi que le nombre de cycles de la simulation.

A l'aide d'un outil de profilage, il serait possible d'obtenir des statistiques sur les parties du code qui requièrent le plus les ressources du processeur. De l'examen de ces statistiques, il serait possible de déduire les parties du code à optimiser dans l'application.

Ces informations seraient très utiles pour le partitionnement au niveau du choix d'affectation d'un module dans la partie logicielle ou matérielle. Grâce au nombre de cycles passés dans chacun des modules logiciels, il serait possible d'identifier le chemin critique qui, comparé aux spécifications, serait d'une grande aide pour déterminer le partitionnement optimal.

7.2.2 Compléter l'API SystemC

L'API SystemC ne comporte pas, pour le moment, toutes les fonctionnalités de SystemC. Elle n'est donc pas totalement compatible. En effet, seules les fonctionnalités élémentaires ont été intégrées permettant pour le moment, le développement d'applications relativement simples. Nous allons parcourir rapidement les différences entre l'API SystemC et la bibliothèque SystemC au niveau des mots clés supportés.

- Les processus de type `SC_METHOD` ne sont pas supportés actuellement. Ils ne jouent pas le rôle de processus perpétuels et ne peuvent donc pas être associés à des tâches pour le RTOS (MicroC/OS-II). Ces processus peuvent être apparentés à de simples fonctions.
- Les évènements ne sont pas supportés. La méthode `wait()` ne permet pas les paramètres de type évènements (sc_event). Cette fonctionnalité n'est pas disponible.
- Les caractéristiques pour la modélisation matérielle comme les types de données sont présentes dans l'API SystemC mais n'ont été testées dans aucun des exemples. Les mots clées `sc_bit, sc_fx, sc_int` ne sont pas supportés.
- Les mécanismes de communication permettant la synchronisation comme les mots clés `sc\_signal, sc\_fifo, sc\_mutex` ne sont pas supportés. En effet, le gestionnaire de communication propose un mécanisme de synchronisation qui pourra être complété de manière à correspondre aux fonctionnalités de ceux proposés par SystemC.
- La bibliothèque SystemC propose la modélisation d'une horloge grâce au mot clé `sc_clock`. Celui-ci n'est pas supporté actuellement et n'a pas de sens dans la partie logicielle.

7.2.3 La gestion de la communication

La gestion des communications peut être optimisée. En effet, l'utilisation des objets fournis par la bibliothèque STL [8] (la librairie standard faisant partie de la norme C++) offre des interfaces et des fonctions utiles permettant de s'affranchir des mécanismes les plus utilisés comme par exemple la manipulation des listes dans le gestionnaire des communications. Cette bibliothèque permet d'éliminer une source d'erreur grâce à la réutilisation de mécanismes dont la fiabilité a été éprouvée. Cependant, l'utilisation des listes est discutable. Elles ne sont pas la solution optimale en terme de rapidité d'exécution et de déterminisme.

Le temps (nombre de cycles) passé dans la communication entre modules fait partie intégrante de la simulation. C'est à dire que lorsque les méthodes d'écriture ou de lecture sont exécutées, celles-ci peuvent durer plusieurs cycles de simulation. Avec la structure composée de listes, plus celles-ci contiennent de messages, plus le traitement est coûteux en terme de temps. C'est ce que nous avons observé lors des tests. Le mécanisme de gestion des messages peut être optimisé afin de réduire dans un premier temps la durée de simulation. Il est également possible d'abstraire le temps de traitement des communications de la simulation. En contrôlant l'exécution de l'ISS, il serait possible de faire exécuter plusieurs instructions dans un seul cycle lors, par exemple, d'une écriture ou d'une lecture. Ainsi, le nombre de cycles passés dans l'API SystemC pourrait être contrôlé et réduit de manière à rendre le temps de communication encore plus proche de la réalité et non dépendant de l'implémentation de la gestion des communications.

Il existe donc plusieurs solutions destinées à améliorer les performances de la communication dans l'API SystemC.

7.3 Les évolutions futures

Le raffinement logiciel décrit le passage d'un langage utilisant des fonctionnalités complexes de haut niveau vers un langage le plus proche possible de l'intégration dans une architecture matérielle (mémoire matériel, SoC). En effet, nous avons vu en introduction les avantages de l'utilisation du langage orienté objet C++. Sa philosophie de réutilisation, ainsi que ses fonctionnalités tendent à simplifier la programmation d'applications complexes souvent constituées d'un très grand nombre de lignes de code. Egalement, nous avons vu que ces principaux avantages ont un prix. La taille et la vitesse d'exécution sont en effet de sérieux critères pour les systèmes sur puce et, de manière plus générale, pour les systèmes embarqués. Il est évident que même si la capacité d'intégration (taille des mémoires) ainsi que la vitesse des processeurs sont en augmentation constante, la quantité de code logiciel ainsi que sa complexité tendent également à augmenter.

Aujourd'hui, le passage d'un langage de haut niveau vers un langage d'intégration est indispensable. Pour le moment, une application décrite en SystemC ne peut pas être intégrée directement sur une architecture. La synthèse logicielle est donc nécessaire et peut être optimisée de manière à rendre cette étape la plus transparente possible pour l'utilisateur, en évitant une étape de re-développement de l'application dans un nouveau langage, ainsi qu'en minimisant l'intervention du développeur. Nous pourrions imaginer un processus de synthèse du logiciel en trois étapes :

1. *L'abstraction du langage SystemC.* Afin d'être intégré dans un environnement ne comportant pas de librairie SystemC, les mots-clés SystemC décrivant l'application doivent disparaître. La notion de module n'a plus lieu d'être, les processus deviennent des tâches propres au RTOS.

2. *L'abstraction des communications.* Une fois le partitionnement effectué, les identités des modules logiciels et matériels sont fixées. Il serait donc possible de parcourir le code source de l'application de manière à remplacer le standard de communication de SPACE (les ports de communications de chacun des modules logiciels ainsi que chacune des fonctions de communication) par d'autres fonctions de communication dépendantes du RTOS utilisé (mécanisme de synchronisation).
3. *L'optimisation du langage.* Cette dernière étape vers la synthèse logicielle pourrait être effectuée de manière automatique par un script dont l'objectif serait de parcourir le code source en éliminant certaines fonctionnalités propres au langage C++ pour les convertir en C destiné à l'implantation finale.

Le développement d'une interface graphique fournirait une aide lors de la conception et de la configuration du système ainsi que pour la visualisation et l'interprétation des résultats. Le projet Picasso [16], développé au GRM, pourrait être le point de départ. Cet outil de codesign est une interface graphique permettant de paramétrer les spécifications d'un système embarqué. Il permettrait l'exploration matérielle et logicielle du futur système de manière graphique. Disposant d'une banque de modules constituant la plate-forme SPACE, l'application décomposée également en modules ou blocs pourrait être connectée au bloc représentant SPACE. Ceci permettrait la génération automatique de la connexion entre les différents modules qui se fait actuellement à la main, dans un fichier séparé (un pour le matériel, un autre pour le logiciel), et qui doit être recréée à chaque changement du partitionnement.

L'architecture utilisée ici n'en est qu'à sa première version. L'intégration de différents types de processeurs comme ceux de type PowerPC ainsi que l'accueil d'autres

RTOS comme par exemple VxWorks font partie des projets envisageables. Le support d'une architecture multiprocesseur, de plus en plus présente dans les SoC, serait également un atout. Par exemple, au niveau L2, il serait intéressant de pouvoir exécuter plusieurs processus logiciels contenant chacun un RTOS. Au niveau L3, il serait possible d'envisager l'utilisation d'un RTOS multiprocesseur.

Nés dans les années 70, les systèmes embarqués font aujourd'hui partie intégrante de notre vie et tendent à se généraliser à tous les domaines où la miniaturisation, la mobilité, ainsi que la puissance de calcul sont nécessaires. Les systèmes embarqués ne sont pas des systèmes ordinaires. Ils requièrent une fiabilité irréprochable du fait du coût élevé de leur fabrication et de leur production de masse. Face à une complexité grandissante, des outils sont indispensables pour leur développement. C'est ce à quoi la méthodologie proposée à travers ce mémoire ainsi que l'outil de partitionnement SPACE, veulent contribuer.

RÉFÉRENCES

[1] ARM, *http ://www.arm.com.* [en ligne]. Page consultée le 14 novembre 2003.

[2] BAKER M., "Co-design made real : Generating and verifying complete system hardware and software implementations". Embedded Systems Conferences. *http ://www.esconline.com/99fallpapers.htm.* [en ligne]. Page consultée le 20 octobre 2003. 1999.

[3] BENNY O., "Implémentation d'un modèle de communication transactionnel dans une plate-forme en systemc". Mémoire de maîtrise. Ecole Polytechnique de Montreal. 2003.

[4] BUTENHOF D. R., *"Programming with POSIX Threads"*. Paperback, 1997.

[5] CHEVALIER J., Thèse en préparation. Ecole Polytechnique de Montréal. 2003.

[6] CHEVALIER J., BENNY O., ABOULHAMID E. M., RONDONNEAU M., BOIS G. et BOYER F. R., "Space : A hardware/software systemc modeling platform including an rtos". Frankfurt, Germany, 2003. Forum on Design Language (FDL03).

[7] Coware N2C System designer, *http ://www.coware.com.* [en ligne]. Page consultée le 2 novembre 2003.

[8] DEITEL and DEITEL, *"C++ How to program"*. Prentice Hall, 2003.

[9] DESMET D., VERKEST D. et MAN H., "Operating system based software generation for systems-on-chip". Proc. Design Automation Conference, (DAC), 2000.

[10] EL-SALAM A.M. et ASHRAF SALEM G. A., *"RTOS Modeling Using SystemC"*. 2002.

[11] GAJSKI D.D., Zhu J., DOMER R., GERSTLAUER A., *"SpecC : Specification Language and Methodology"*. Norwell, MA : Kluwer Academic Publishers, 2000.

[12] GDB, *http ://sources.redhat.com/gdb.* [en ligne]. Page consultée le 24 octobre 2003.

[13] GERSTLAUER A., YU H. et GAJSKI D.D., "Rtos modeling for system level design". Design, Automation and Test in Europe Conference and Exhibition, 2003.

[14] GNU, *http ://www.gnu.com.* [en ligne]. Page consultée le 22 novembre 2003.

[15] GROTKER L., LIAO S., MARTIN G. et SWAN S., *"System Design with SystemC"*. Kluwer Academic Publishers, 2002.

[16] HENEAULT Y., BOIS G., ABOULHAMID M., BAILLAIRGE J., et YOUSEFPOUR P., "Picasso : A h/s capture tool based on vsia recommendations". Proc. of International Workshop On IP Based Synthesis and System Design, France 1999.

[17] HERRERA P., POSADAS H., SANCHEZ P. et VILLAR E., "Systematic embedded software generation from systemc". pages 142–147, Munich, Germany, 2002. Proceeding of Design, Automation and Test in Europe Conference and Exhibition (DATE03).

[18] Insight., *http ://sources.redhat.com/insight.* [en ligne]. Page consultée le 5 juin 2003.

[19] KarbonKernel, *http ://savannah.nongnu.org/projects/carbonkernel/.* [en ligne]. Page consultée le 8 septembre 2003.

[20] KEPPEL D., "Tools and techniques for building fast portable threads packages". University of Washington, Technical Report UWCSE 93-05-06. 1993.

[21] LABROSSE J., *"MicroC/OS-II, The Real-Time Kernel, Second Edition"*. CMP Books, 2002.

[22] Mentor Graphics, Seamless user's and reference manual. software version 4.0.(logiciel). 2000.

[23] NICOLESCU G., *"Spécification et validation des systèmes hétérogènes embarqués"*. Mémoire de thèse. 2002.

[24] NICOLESCU G., YOO S., OUCHHIMA A., et JERRAYA A.A., *"Validation in Component-Based Design Approach for Multicore SoCs"*. ACM Press, 2002.

[25] OSCI, *"SystemC Version 2.0.1 User's Guide"*. http ://www.systemc.org, 2002. [en ligne]. Page consultée le 3 mars 2002.

[26] OSCI Synopsys, "Functional specification for SystemC 2.0.1". *http ://www.systemc.org.* [en ligne]. Page consultée le 5 mars 2002. 2002.

[27] OsKit, *http ://www.cs.utah.edu/flux/oskit/index.html.* [en ligne]. Page consultée le 16 juin 2003.

[28] SCHIRRMEISTER F., "Integration platforme based design using cierto virtual component co-design(vcc)". Cadence VCC White Paper and Overview for Customers. *http ://www.cadence.com.* [en ligne]. Page consultée le 10 octobre 2003. 1999.

[29] VERILOG, *http ://www.verilog.com.* [en ligne]. Page consultée le 6 mars 2002.

[30] VHDL, *http ://www.vhdl.com.* [en ligne]. Page consultée le 5 février 2003.

[31] VxSim, *http ://www.windriver.com/products/VxSim.* [en ligne]. Page consultée le 6 septembre 2003.

[32] VxWorks, *http ://www.windriver.com/products/vxworks5.* [en ligne]. Page consultée le 6 septembre 2003.

[33] WindRiver, *http ://www.windriver.com.* [en ligne]. Page consultée le 3 décembre 2002.

[34] YOO S. et JERRAYA A.A., "Introduction to hardware abstraction layers for soc". Munich, Germany, 2003. System-Level Synthesis Group (DATE03).

ANNEXE I

L'API SYSTEMC

I.1 La structure de l'API SystemC

La structure de l'API SystemC est très similaire à celle de la bibliothèque SystemC puisqu'elle a été développée à partir de celle-ci. En effet, l'API SystemC est en fait la bibliothèque SystemC sans son simulateur. Celui-ci a été remplacé par les appels de fonctions des services du RTOS utilisé (actuellement MicroC/OS-II). Le principal avantage était de pouvoir évoluer facilement vers une nouvelle version de la bibliothèque SystemC qui aurait pu remplacer, sans trop d'effort, la version actuelle. Ce principe aurait permis d'éviter de modifier le code source de l'API SystemC afin de rajouter de nouvelles fonctionnalités correspondantes aux évolutions futures de la bibliothèque SystemC. Cependant, la structure influence les performances au niveau de la simulation logicielle. En effet, la bibliothèque SystemC renferme un grand nombre de fonctionnalités qui peuvent être simplifiées du fait de l'utilisation d'un ordonnancement externe à la bibliothèque et des spécifications du mode de communication propre à l'architecture SPACE. Par exemple le port unique de communication de SPACE hérite du port de communication standard de la bibliothèque SystemC. Cela implique, lors de la simulation, tout un parcourt imbriquant une série d'appels de fonctions pour enfin exécuter celle parmi l'interface de communication standard à SPACE. Ce dispositif pourrait être simplifié pour appeler directement la fonction correspondante, sans passer par le mécanisme de gestion des ports de communication de SystemC. Cela permettrait de diminuer le nombre d'appels de fonctions imbriquées et ainsi, augmenterait la rapidité

d'exécution de l'application. L'API SystemC mérite une restructuration dans un souci d'optimisation de manière à réduire la quantité de code qu'elle renferme ralentissant la simulation. La fonctionnalité a été démontrée. Des efforts doivent être portés sur l'optimisation de manière à obtenir des résultats de simulation plus performants.

Le gestionnaire de communication présent dans l'API SystemC est constitué d'une série de listes construites à partir de la bibliothèque S.T.L. (Standard Template Library). La méthode employée n'est pas déterministe et peu performante. Plus il y a de modules, et plus la communication est ralentie par le parcourt des différentes listes pour écrire ou lire un message. Cette méthode pourrait être optimisée par exemple en s'inspirant du mécanisme utilisé par MicroC/OS-II pour rechercher une tâche de plus grande priorité dans un tableau à 2 dimensions. Les méthodes de lecture et d'écriture en mode bloquant ou non bloquant vers un module ou un périphérique comme par exemple la mémoire, sont regroupées dans le fichier `api_softcom.cpp` dans le répertoire `communication` tandis que la gestion des différentes listes est regroupée dans le fichier `com.cpp` du répertoire `kernel` .

I.2 Les caractéristiques

Tous les modules des applications créés pour être exécutés sur l'architecture SPACE au niveau 3 ou au niveau 2 héritent d'une classe de base appelée `space_base_module`. Celle-ci comprend le port de communication ainsi que des interfaces disponibles permettant de simplifier l'utilisation du canal de communication.

Comme nous l'avons vu dans le chapitre présentant l'API SystemC, l'attribution des priorités est effectuée par l'affectation des identifiants des modules. Il n'est donc pas possible d'avoir des modules de même identifiant et donc des processus

de même priorité. Un module ne peut pas contenir plus d'un processus puisque le mécanisme de communication ne pourrait actuellement pas distinguer les messages destinés à l'un plutôt qu'à l'autre.

La mise en attente d'un processus est effectuée par l'appel de méthode `wait()`. Cette méthode de la bibliothèque SystemC a été simplifiée, elle ne peut prendre que deux types de paramètres : un délais exprimé en unité de temps ou un nombre de cycle d'horloges. Cette convention permet de garder une compatibilité dans la fonctionnalité du module, qu'il soit dans la partie logicielle ou matérielle.

La gestion des évènements n'est actuellement pas implémenté. Il n'est donc pas possible de mettre un processus en attente sur un événement, à moins d'utiliser explicitement une communication bloquante. Le mécanisme de gestion des événements pourra dans l'avenir s'inspirer de la communication bloquante du gestionnaire de communication. Comme nous l'avons vu précédemment, l'utilisation de plusieurs processus à l'intérieur d'un seul module n'est actuellement pas permis, la synchronisation devra se faire entre module en utilisant l'interface de communication.

Les signaux ne sont également pas implémentés pour les mêmes raisons que les évènements décrits dans le paragraphe précédent. La propagation d'un signal entre module correspond à une communication qui passera par le port spécifique à l'architecture SPACE. Cette communication se fera par l'intermédiaire des méthodes de lecture et d'écriture. Ce mécanisme pourra être encapsulé afin d'être transparent afin de rendre l'utilisation des signaux valide dans la description d'applications destinées à êtres simulés sur l'architecture SPACE.

I.3 Le changement du RTOS

Le changement de MicroC/OS-II par un autre RTOS implique plusieurs modifications mineures dans le code de l'API SystemC. Cette procédure n'est actuellement pas automatisée, cependant la plupart des modifications sont concentrées dans une partie de l'API et ne nécessitent pas le parcourt de tout le code source de celle-ci. La création d'un processus, sa mise en attente, l'utilisation de mécanismes de synchronisation sont des fonctions dont les appels sont propres au RTOS utilisé. Toutes ces fonctions se retrouvent dans les fichiers `api_cor_ucosii.c` et `api_cor_ucosii.h`. Certaines de ces fonctions sont utilisées par le gestionnaire de communication et d'autres par la procédure d'initialisation de l'API SystemC. Elles encapsulent les méthodes du RTOS (ici, MicroC/OS-II). L'utilisation d'un nouveau RTOS implique donc la création de fonctions semblables encapsulant les services du RTOS utilisé. L'attribution des valeurs des paramètres des différentes fonctions est pour le moment laissé au soin du développeur. Ces paramètres varient d'un RTOS à l'autre. Par exemple, les paramètres lors de la création d'une tâche MicroC/OS-II sont différents de ceux lors de la création d'une tâche VxWorks. Certains de ces paramètres peuvent être regroupés dans un fichier de configuration comme par exemple la taille des piles des différentes tâches. La création des piles des différents processus ne s'effectue pas de manière dynamique. Actuellement, un tableau à deux dimensions conserve les piles de 10 tâches. La construction des piles s'effectue de manière statique dans MicroC/OS-II. Ce mécanisme pourra être automatisé.

ANNEXE II

L'INTÉGRATION D'UN OS POUR SPACE

A travers ce chapitre nous allons détailler les caractéristiques permettant d'adapter MicroC/OS-II afin qu'il puisse fonctionner sur le processeur de type ARM présent sur l'architecture SPACE sous forme d'ISS. Cette adaptation qui requiert des fonctionnalités écrites la plupart du temps en C ou en assembleur, propres au processeur, s'appelle un port.

La figure suivante présente l'architecture de MicroC/OS-II qui est structurée en trois groupes : la configuration, les fonctions indépendantes du processeurs et disponibles pour le développement d'applications et enfin le port qui comprends les fonctions spécifiques au processeur utilisé.

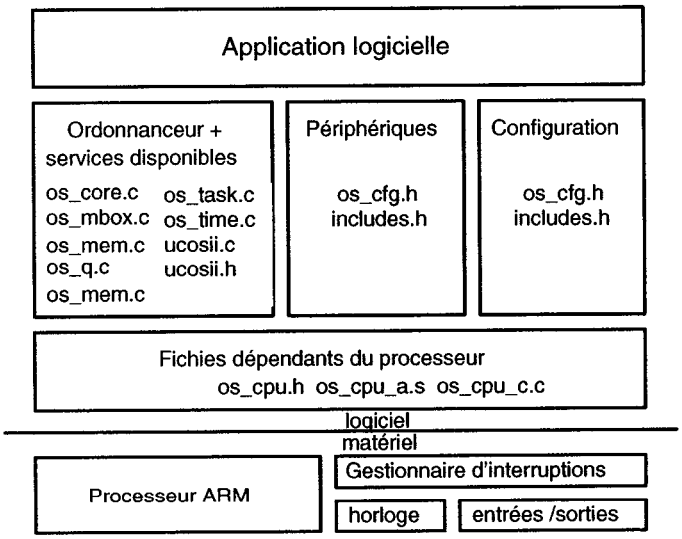

FIGURE II.1 Architecture MicroC/OS-II

Nous allons parcourir chacun des différents groupes afin de mieux comprendre leur fonctionnalité.

Les fichiers de configurations : Ceux-ci permettent de paramétrer le RTOS pour lui donner un certain nombre de caractéristiques. Ces caractéristiques devront correspondre au besoin de l'application. Il est donc possible de réduire la taille de MicroC/OS-II en dégageant toutes les fonctionnalités non utilisées. Rappelons que la taille est un facteur important dans le développement des systèmes embarqués, c'est pourquoi cette caractéristique se retrouve dans beaucoup de RTOS commerciaux tels uClinux, eCOS, VxWorks, et même ceux disponibles au grand publique comme Linux.

Les fichiers spécifiques au processeur : Ces fichiers jouent un rôle déterminant puisqu'ils font le lien entre le processeur et le reste du RTOS. Nous allons en détailler les principales fonctionnalités. Tout d'abord, la portabilité d'un RTOS est très importante au niveau commercial, c'est pourquoi, une certaine politique au niveau de la structure du code doit être mise en place afin de faciliter cette opération de passage d'un processeur à l'autre afin de suivre l'évolution du marché. Concernant le code de Micro/C-II, tous les types de données ont été redéfinis et regroupés dans un seul et même fichier (OS_CPU.H). Les processeurs ayant des tailles de mots différents, il sera facile de modifier un seul et même fichier plutôt que de parcourir tout le code source du noyau. Il en va de même pour toutes les fonctions spécifiques au processeur simplifiant ainsi la portabilité du RTOS. Concernant le compilateur, celui-ci gère les même tailles de mots pour les processeurs de type Intel et le type de processeur ARM utilisé sur SPACE.

La gestion de l'arrivée des interruptions est très importante et vitale pour tous systèmes d'exploitation. La désactivation des interruptions permet d'accéder à une section dite critique pendant laquelle le processeur ne doit pas se faire interrompre. Durant la désactivation, le code critique est protégé d'une interruption de changement de contexte ou de tout autre interruption (Interrupt Service Routine). MicroC/OS-II fournit deux fonctions pour gérer l'activation et la désactivation

des interruptions. La fonction OS_ENTER_CRITICAL() permettra de désactiver les interruptions en pointant sur la fonction suivante :

```
ARMDisableInt:
        MRS     r12, CPSR
        ORR     r12, r12, #NoInt      ; NoInt = 0x80
        MSR     CPSR, r12
        MOV    pc,lr
```

Tandis que l'activation des interruptions se fera par la fonction OS_EXIT_CRITICAL() correspondant aux instructions suivantes :

```
ARMEnableInt:
        MRS     r12, CPSR
        BIC       r12, r12, #NoInt
        MSR     CPSR, r12
        MOV    pc,lr
```

Il existe plusieurs méthodes pour implémenter cette gestion. La méthode employée précédemment ne tient pas en compte l'état des interruptions avant l'appel de la fonction. C'est à dire que si, par exemple, une fonction MicroC/OS-II est encapsulée dans une section critique, les interruptions seront désactivées au retour de cette fonction. Le reste des instructions suivantes ne seront plus protégées par la section critique puisque l'état des interruptions avant chaque désactivation n'est jamais pris en compte. Une autre méthode consiste à conserver l'état des interruptions lors d'une désactivation et de restaurer cet état lors de la réactivation. Cette méthode tend à augmenter la latence des interruptions c'est à dire, le temps entre lequel une interruption survient et celui ou le processeur commence à sauvegarder le contexte

de la tâche en cours. En effet, la sauvegarde de l'état des interruptions requiert des instructions et donc du temps processeur (cycles) supplémentaire. Les deux méthodes sont valables, tout dépendemment de la fonctionnalité que l'on veut privilégier.

Le démarrage de l'ordonnanceur (OSStartHighRdy) est également dépendant du processeur. Rappelons que cette fonction permet d'exécuter la tâche de plus grande priorité. Les registres du processeur, initialisés et contenus dans la pile de la tâche de plus grande priorité (pointée par OSTCBHighRdy) sont présentés à la figure II.2.

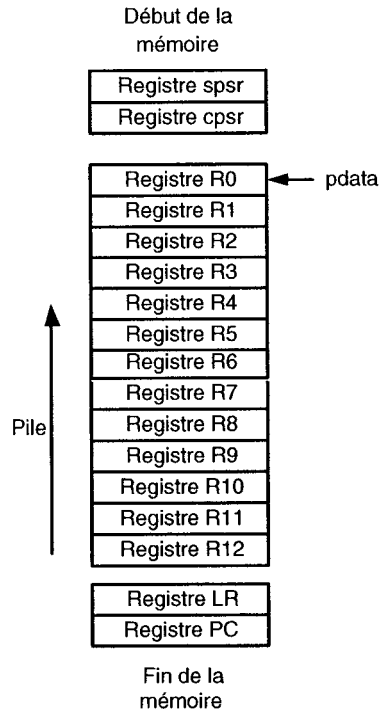

FIGURE II.2 Etat de la pile lors de la création d'une tâche

La pile est très importante et dépend également des caractéristiques du processeur. En effet, sur certains processeurs, le pointeur de pile est incrémenté lors de l'exécution de l'instruction d'empilement, pour d'autre, celui-ci se décrémente. Dans MicroC/OS-II, une macro permet de prendre en considération cette caractéristique

sans modifier le mécanisme d'initialisation de la pile de chacune des tâches du RTOS. Ici, OS_STK_GROWTH est fixé à 1 pour indiquer que le pointeur de pile décrémente son adresse.

Le changement de contexte est une fonction dont le mécanisme est dépendant du processeur. Celle-ci permet la mise en attente de la tâche en cours d'exécution pour la remplacer par une tâche prête à être exécutée. Cette fonction peut être déclenchée par une interruption ou de manière volontaire par la tâche en cours d'exécution, en simulant une interruption (logicielle). Les fonctions respectives OSIntCtxSw et OSCtxSw sont les fonctions de changement de contextes, elles sont donc dépendantes du processeur. Concernant MicroC/OS-II, ces fonctions sont totalement écrites en assembleur. Leur rôle consiste à sauvegarder les registres du processeur de la tâche en cours d'exécution dans sa pile, puis de remplacer leur contenu par les valeurs sauvegardées dans la pile de la tâche à exécuter.

La routine d'interruption de l'horloge système (OSTickISR) permet de synchroniser l'ordonnancement ainsi que la gestion des délais d'attente des tâches. La fréquence d'apparition de ces interruptions est fixée à 2Hz (une interruption toutes les 500ms) pour le niveau L3 où le RTOS est exécuté sur l'ISS de d'architecture SPACE. Rappelons que cette fréquence se trouve entre 10 et 100Hz selon la littérature. Celle-ci pourra donc être ajustée à des fins d'optimisation.

Les fichiers pour le développement des applications : Ces fichiers regroupent l'ensemble des fonctionnalités que fournit MicroC/OS-II pour le développement d'applications. Parmi elles, nous retrouvons des fonctions de synchronisation pour la communication entre les tâches, comme les boîtes de messages ou les files d'attentes, et des fonctions de gestion du temps. Ces fonctions utilisent des mécanismes qui permettent de mettre une tâche en attente d'un événement ou d'un temps donné. Nous n'allons pas détailler toutes les fonctionnalités qui sont regroupées dans le

manuel de MicroC/OS-II en référence.

L'initialisation des périphériques pour la simulation au niveau 3 (SPACE) : Plusieurs méthodes ont été développées afin d'initialiser les périphériques comme l'horloge et le gestionnaire d'interruption. Certaines d'entre elles permettent d'initialiser et de démarrer l'horloge, qui joue le rôle de minuterie en déclenchant une interruption selon une période qui elle aussi peut être initialisée par ces méthodes. D'autres méthodes sont chargées d'initialiser le gestionnaire d'interruption afin d'activer son fonctionnement et d'acquitter une ou plusieurs interruptions. Il existe actuellement deux interruptions possibles : la minuterie et la communication matérielle. Il n'existe aucune fonction permettant de rajouter de manière automatique d'autres types d'interruptions. Ceci n'a pas été jugé pertinent pour démontrer la fonctionnalité de la méthodologie et pourra faire l'objet d'évolutions futures. Ces différents modules matériels, contrôlés par ces fonctions peuvent fournir des informations de déverminage dépendamment de l'état de leurs registres. Une autre méthode permet d'aiguiller l'affichage vers le module de mémoire vidéo présent dans l'architecture de SPACE. Cette mémoire permet de simuler une véritable mémoire vidéo, toutes les données envoyées vers ce module sont affichées à l'écran. Ceci a pour objectif de centraliser l'affichage en ajoutant des informations permettant de distinguer l'origine du message, qu'il soit en provenance d'un module matériel, logiciel, une information de déverminage ou concernant les résultats de simulation.